VisualDSP++® 4.0 User's Guide

Revision 1.0, January 2005

Part Number
82-000420-02

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

© 2005 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, the CROSSCORE logo, Blackfin, SHARC, TigerSHARC, EZ-KIT Lite, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

Apex-ICE and Summit-ICE are trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual xxiii
Intended Audience xxiii
Manual Contents xxiv
What's New in This Manual xxv
Technical or Customer Support xxvi
Supported Processors xxvii
Product Information xxviii
MyAnalog.com xxviii
Processor Product Information xxix
Related Documents xxix
Online Technical Documentation xxx
Accessing Documentation From VisualDSP++ xxxi
Accessing Documentation From Windows xxxi
Accessing Documentation From the Web xxxii
Printed Manuals xxxii
VisualDSP++ Documentation Set xxxii
Hardware Tools Manuals xxxiii
Processor Manuals xxxiii

Data Sheets xxxiii
Notation Conventions xxxiii

INTRODUCTION TO VISUALDSP++

VisualDSP++ Features 1-2
Integrated Development and Debugging 1-2
Code Development Tools 1-2
Source File Editing Features 1-3
Project Management Features 1-4
Debugging Features 1-5
VDK Features 1-6
VisualDSP++ 4.0 Features 1-7
License Management 1-9
Licensing Options 1-10
License Status 1-11
Temporary Licenses 1-11
Valid Versus Expired Licenses 1-11
Client Licenses 1-11
License Installation 1-12
Installing a License Shipped With an EZ-KIT Lite Evaluation System 1-12
VisualDSP++ Product Upgrades 1-13
Product Serial Numbers 1-13
Project Development 1-14
Overview of Programming With VisualDSP++ 1-14

Project Development Stages 1-17
Simulation 1-17
Evaluation 1-18
Emulation 1-18
Targets 1-18
Simulation Targets 1-18
EZ-KIT Lite Targets 1-19
Emulation Targets 1-19
Platforms 1-19
Hardware Simulation 1-20
Debugging Overview 1-20
VisualDSP++ Kernel 1-22
Program Development Steps 1-22
Step 1: Create a Project 1-23
Step 2: Configure Project Options 1-23
Step 3: Add and Edit Project Source Files 1-23
Adding Files to Your Project 1-23
Creating Files to Add to Your Project 1-24
Editing Files 1-24
Managing Project Dependencies 1-24
Step 4: Specifying Project Build Options 1-24
Configuration 1-25
Project-Wide File and Tool Options 1-25
Individual File and Tool Options 1-25

Step 5: Build a Debug Version of the Project 1-26
Step 6: Create a Debug Session and Load the Executable ... 1-26
Step 7: Run and Debug the Program 1-26
Step 8: Build a Release Version of the Project 1-26
Code Development Tools 1-27
Compiler 1-28
C++ Run-Time Libraries 1-29
Assembler 1-30
Linker 1-31
Expert Linker 1-34
Expert Linker Window 1-35
Memory Map Pane Right-Click Menu 1-36
Stack and Heap Usage 1-39
Archiver 1-41
Splitter 1-41
Loader 1-42
Processor Projects 1-44
Project Options 1-45
Project Groups 1-46
Project Group Files 1-47
Source Code Control (SCC) 1-48
Makefiles 1-49
Rules 1-50
Output Window 1-51

Example Makefile 1-51
Project Configurations 1-53
Customized Project Configurations 1-54
Project Build 1-54
Build Options 1-55
File Building 1-56
Batch Builds 1-56
Pre-Build and Post-Build Options 1-56
Command Syntax 1-57
Project Dependencies 1-57
Project Window Rules 1-57
VisualDSP++ Help System 1-59

ENVIRONMENT

Parts of the User Interface 2-2
Title Bar 2-3
Additional Information in Title Bars 2-4
Title Bar Right-Click Menus 2-4
Control Menu 2-5
Program Icons 2-5
Editor Windows 2-6
Debugging Windows 2-6
Menu Bar 2-6
Toolbars and User Tools 2-7
Built-In Toolbars 2-7

Toolbar Customization 2-8
Toolbars: Docked vs. Floating 2-8
Toolbar Button Appearance 2-9
Toolbar Shape 2-11
Toolbar Rules 2-11
User Tools 2-12
Status Bar 2-12
VisualDSP++ Windows 2-14
Project Window 2-14
Project View 2-15
Project Dependencies 2-16
Project Nodes 2-17
Project Page Right-Click Menus 2-18
Project Group Icon Right-Click Menu 2-18
Project Icon Right-Click Menu 2-19
Folder Icon Right-Click Menu 2-20
File Icon Right-Click Menu 2-20
Project Folders 2-21
Project Files 2-22
Project Window Icons for Source Code Control (SCC) 2-23
File Associations 2-24
Automatic File Placement 2-24
File Placement Rules 2-25
Example 2-25

Kernel Page 2-26
Editor Windows 2-28
Editor Window Symbols 2-30
Bookmarks 2-30
Syntax Coloring 2-30
Viewing Modes: Source Mode vs. Mixed Mode 2-31
Source Mode 2-31
Mixed Mode 2-32
Editor Tab Mode 2-33
Context-Sensitive Expression Evaluation 2-34
Viewing an Expression 2-34
Highlighting an Expression 2-34
Right-Click Menu 2-35
Output Window 2-36
Viewing Error Message Details 2-36
Output Window Tabs 2-38
Build Page 2-38
Console Page 2-38
Output Window Error Messages 2-39
Error Message Severity Hierarchy 2-40
Syntax of Help for Error Messages 2-40
Promoting, Demoting, and Suppressing Error Messages . 2-42
Log File 2-45
Output Window Customization 2-46

Right-Click Menu 2-47
Script Command Output 2-48
Debugging Windows 2-50
Disassembly Windows 2-52
Other Disassembly Window Features 2-54
Right-Click Menu 2-55
Disassembly Window Symbols 2-56
Expressions Window 2-57
Right-Click Menu 2-58
Expressions Permitted in an Expression Window 2-59
Trace Windows 2-60
Locals Window 2-62
Statistical/Linear Profiling Results Window 2-63
Window Components 2-63
Left Pane 2-63
Right Pane 2-65
Status Bar 2-65
Right-Click Menu 2-65
Window Operations 2-66
Changing the Window View 2-66
Displaying a Source File 2-67
Displaying Functions in Libraries 2-67
Working With Ranges 2-68
Switching Display Modes 2-69

Filtering PC Samples With No Debug Information 2-71
Call Stack Window 2-72
Memory Windows 2-72
Number Formats in Memory Windows 2-73
Right-Click Menu 2-75
Expression Tracking in a Memory Window 2-76
Memory Window Display Customization 2-78
Background Telemetry Channels (BTCs) 2-78
BTC Definitions in Your Program 2-79
Enabling BTC on ADSP-2126x and ADSP-BF36x Processors 2-80
BTC Priority 2-81
BTC Memory Window 2-81
Right-Click Menu 2-84
Memory Map Windows 2-85
Register Windows 2-86
Stack Windows 2-89
Custom Registers Windows 2-89
Multiprocessor Window 2-90
Multiprocessor Window Pages 2-91
Status Page 2-91
Groups Page 2-92
Operating on Multiprocessor Groups 2-92
Focus 2-93
Right-Click Menu 2-93

Pipeline Viewer Window 2-94
Right-Click Menu 2-95
Pipeline Viewer Properties Dialog Box 2-96
Pipeline Viewer Window Event Icons 2-97
Pipeline Instruction Event Details 2-98
Cache Viewer Window 2-99
Configuration Page 2-102
Detailed View Page 2-103
History Page 2-104
Performance Page 2-106
Histogram Page 2-107
Address View Page 2-108
VDK Status Window 2-109
VDK State History Window 2-111
Thread Status and Event Colors 2-112
Window Operations 2-113
Right-Click Menu 2-113
Target Load Window 2-114
Plot Windows 2-115
Plot Window Features 2-116
Status Bar 2-116
Tool Bar 2-117
Right-Click Menu 2-118
Plot Window Statistics 2-120

Plot Configuration 2-121
Plot Window Presentation 2-122
Plot Presentation Options 2-124
Image Viewer 2-124
Automation Interface 2-126
Toolbar 2-126
Status Bar 2-127
Right-Click Menu 2-127

DEBUGGING

Debug Sessions 3-2
Debug Session Management 3-3
Simulation vs. Emulation 3-3
Breakpoints 3-3
Watchpoints 3-4
Multiprocessor (MP) System Debugging 3-4
Setting Up a Multiprocessor Debug Session 3-4
Debugging a Multiprocessor System 3-5
Focus and Pinning 3-6
Window Title Bar Information 3-6
Additional Focus Indication 3-7
Code Analysis Tools 3-7
Statistical Profiles and Linear Profiles 3-8
Simulation: Linear Profiling 3-8
Emulation: Statistical Profiling 3-8

Traces 3-9
Program Execution Operations 3-10
Selecting a New Debug Session at Startup 3-10
Loading the Executable Program 3-11
Program Execution Commands 3-11
Restarting the Program 3-12
Performing a Restart During Simulation 3-12
Performing a Restart During Emulation 3-13
Breakpoints 3-13
Unconditional and Conditional Breakpoints 3-14
Watchpoints 3-14
Hardware Breakpoints 3-14
Latency 3-15
Restrictions 3-15
Simulation Tools 3-16
Interrupts 3-16
Input/Output Simulation (Data Streams) 3-16
Plots 3-18
Plot Types 3-19
Line Plots 3-20
X-Y Plots 3-21
Constellation Plots 3-22
Eye Diagrams 3-23
Waterfall Plots 3-24

Spectrogram Plots 3-26
Flash Programmer 3-27
Flash Devices 3-27
Flash Programmer Functions 3-27
Flash Driver 3-28
Flash Programmer Window 3-29

REFERENCE INFORMATION

Glossary A-2
File Types A-24
Toolbar Buttons A-27
Keyboard Shortcuts A-32
Working With Files A-32
Moving Within a File A-33
Cutting, Copying, Pasting, Moving Text A-34
Selecting Text Within a File A-34
Working With Bookmarks in an Editor Window A-35
Building Projects A-36
Using Keyboard Shortcuts for Program Execution A-36
Working With Breakpoints A-37
Obtaining Online Help A-37
Miscellaneous A-37
IDDE Command-Line Parameters A-38
Extensive Scripting A-39
Text Operations A-43

Regular Expressions vs. Normal Searches A-43
Specific Special Characters A-44
Special Rules for Sequences A-45
Repetition and Combination Characters A-45
Match Rules A-46
Tagged Expressions in Replace Operations A-46
Comment Start and Stop Strings A-47
Online Documentation A-48
Printing Online Documentation A-48
Viewing Online Help A-50
Online Help A-51
Help Window A-51
Context-Sensitive Help A-52
Viewing Menu, Toolbar, or Window Help A-54
Viewing Dialog Box Help A-54
Viewing Window Help A-55
Copying Example Code From Help A-55
Printing Help A-56
Bookmarking Frequently Used Help Topics A-56
Navigating in Online Help A-57
Searching Help A-59
Full-Text Searches A-59
Rules for Full-Text Searches A-60
Advanced Search Techniques A-61

Wildcard Expressions A-61
Boolean Operators A-63
Nested Expressions A-64
Rules for Advanced Searches A-64
About VisualDSP++ Dialog Box A-65
Window Operations A-70
Window Manipulation A-70
Right-Click Menu Options A-70
Scroll Bars and Resize Pull-Tab A-71
Windows: Docked vs. Floating A-71
Docked Windows A-72
Floating Windows A-73
Window Position Rules A-74
Standard Windows Buttons A-75

SIMULATION OF SHARC PROCESSORS

Anomaly Options B-2
ADSP-21x6x Processor Anomalies B-2
Shadow Write FIFO Anomaly (ADSP-2116x Only) B-2
SIMD Read from Internal Memory With Shadow Write FIFO Hit Anomaly (ADSP-2116x Only) B-3
Event Options B-4
FP Denorm B-4
Short Word Anomaly B-4

Access to ADSP-21065L Short-Word Internal Memory 9th Column at Even Addresses B-7
Recording a Simulator Anomaly or Event B-7
Select Processor ID Options B-10
Simulator Options B-10
No Boot Mode B-10
Load Sim Loader Options B-11
SPI Simulation in Slave Mode B-13

SIMULATION OF TIGERSHARC PROCESSORS

ADSP-TS101 Processors C-1
Simulator Timing Analysis Overview C-2
Pipeline Stages C-2
Stalls C-3
Stalls Due to IALU Dependency C-4
Stalls Due to Compute Block Dependency C-5
Aborts C-6
Aborts Due to an Unpredicted Change of Flow C-6
Abort Due to Mispredicted Change of Flow C-7
Branch Target Buffer Hits C-8
Pipeline Viewer and Disassembly Window Operations C-8
Current Program Counter Value C-9
Stepping C-11
Simulator Options C-12
ADSP-TS20x Processors C-13

Simulator Timing Analysis Overview C-13
Pipeline Stages C-14
Stalls C-15
Stalls Due to IALU Dependency C-15
Stalls Due to Compute Block Dependency C-16
Stalls Due to a Cache Miss C-17
Aborts C-17
Aborts Due to an Unpredicted Change of Flow C-18
Abort Due to Mispredicted Change of Flow C-19
Branch Target Buffer Hits C-20
Pipeline Viewer and Disassembly Window Operations C-20
Current Program Counter Value C-21
Stepping C-23
Simulator Options C-23

SIMULATION OF BLACKFIN PROCESSORS

Peripheral Support in Simulators D-2
Special Considerations for Peripherals D-7
Universal Asynchronous Receiver/Transmitter Peripheral D-7
Timer (TMR) Peripheral D-7
Simulator Instruction Timing Analysis for ADSP-BF535 Processors D-9
Stall Reasons D-9
Kill Reasons D-10
Pipeline Viewer Window Examples D-11
Pipeline Viewer Window Messages D-12

Pipeline Viewer Detail View Stall Event Messages D-12
Kills Detected Messages D-16
Multicycle Instructions D-17
Abbreviations in Pipeline Viewer Messages D-17
Simulator Instruction Timing Analysis for ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 Processors D-19
Stall Reasons D-19
Kill Reasons D-20
Pipeline Viewer Window Examples D-20
Multicycle Instructions and Latencies D-22
Multicycle Instructions D-22
Push Multiple or Pop Multiple D-22
32-Bit Multiply (modulo 232) D-23
Call and Jump D-23
Conditional Branch D-23
Return D-24
Core and System Synchronization D-24
Linkage D-25
Interrupts and Emulation D-25
TESTSET D-25
Instruction Latencies D-26
Accumulator to Data Register Latencies D-27
Register Move Latencies D-28
Move Conditional and Move CC Latencies D-30
Loop Setup Latencies D-31

Instructions Within Hardware Loop Latencies D-32
Instruction Alignment Unit Empty Latencies D-33
L1 Data Memory Stalls D-34
Minibank Access Collision D-35
SRAM Access (1-Cycle Stall) D-35
Cache Access (1-Cycle Stall) D-36
MMR Access D-39
System Minibank Access Collision D-39
Store Buffer Overflow D-39
Store Buffer Load Collision D-40
Load/Store Size Mismatch D-40
Store Data Not Ready D-41
Instruction Groups D-41
Register Groups D-42
Compiled Simulation D-43
Specifying a Session for Compiled Simulation D-43

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for digital signal processing (DSP) applications.

Purpose of This Manual

The *VisualDSP++ 4.0 User's Guide* describes the features, components, and functions of VisualDSP++. Use this guide as a reference for developing programs for SHARC®, TigerSHARC®, and Blackfin® processors.

This manual does not include detailed procedures for building and debugging projects. For how-to information, refer to VisualDSP++ online Help and the *VisualDSP++ 4.0 Getting Started Guide*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents

The manual consists of:

- Chapter 1, "Introduction to VisualDSP++"
  Describes VisualDSP++ features, license management, project development, code development tools, VCSE, and DSP projects
- Chapter 2, "Environment"
  Focuses on window features, operations, and customization for the main window and debugging windows.
- Chapter 3, "Debugging"
  Describes debug sessions, code analysis tools, program execution operations, simulation tools, Image Viewer, plots, and Flash Programmer.
- Appendix A, "Reference Information"
  Describes file types, keyboard shortcuts, command-line parameters, scripting, toolbar buttons, and text operations; also provides a glossary and describes online Help features and operations.
- Appendix B, "Simulation of SHARC Processors"
  Describes the simulator options available on the **Anomalies**, **Events**, **Simulator**, **Load Sim Loader**, and **Select Processor ID** submenus under **Settings**; also explains how to record simulator anomalies and events, and describes SPI simulation in slave mode.

- Appendix C, "Simulation of TigerSHARC Processors"
  Describes simulator instruction timing analysis, pipeline stages, the Pipeline Viewer, stalls, aborts, the current program counter value, stepping, and the **Select Loader Program** command on the **Simulator** submenu under **Settings**.
- Appendix D, "Simulation of Blackfin Processors"
  Provides an overview of peripheral support for Blackfin simulators and describes limitations of the simulation software models, simulator instruction timing analysis, and compiled simulation.

What's New in This Manual

The *VisualDSP++ 4.0 User's Guide* supports all Analog Devices, Inc. processor families listed in "Supported Processors".

Refer to the *VisualDSP++ 4.0 Product Release Bulletin* for information on all new and updated VisualDSP++® 4.0 features and other release information.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to
  `dsptools.support@analog.com`
- E-mail processor questions to
  `embedded.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++ 4.0.

TigerSHARC (ADSP-TSxxx) Processors
The name "TigerSHARC" refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC processors:

ADSP-TS101	ADSP-TS201	ADSP-TS202	ADSP-TS203

SHARC (ADSP-21xxx) Processors
The name "SHARC" refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC processors:

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363
ADSP-21364	ADSP-21365	ADSP-21366	ADSP-21367
ADSP-21368	ADSP-21369		

Blackfin (ADSP-BFxxx) Processors
The name "Blackfin" refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin processors:

ADSP-BF531	ADSP-BF532 (formerly ADSP-21532)
ADSP-BF533	ADSP-BF535 (formerly ADSP-21535)
ADSP-BF561	AD6532

ADSP-BF534	ADSP-BF536
ADSP-BF537	ADSP-BF566
ADSP-BF538	ADSP-BF539

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

`MyAnalog.com` is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Registration

Visit `www.myanalog.com` to sign up. Click **Register** to use `MyAnalog.com`. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at `www.analog.com/processors`, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to `embedded.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  089/76 903-557 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` or `ftp 137.71.23.21`
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ 4.0 Getting Started Guide*
- *VisualDSP++ 4.0 Assembler and Preprocessor Manual*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*

- *VisualDSP++ 4.0 Linker and Utilities Manual*
- *VisualDSP++ 4.0 Loader Manual*
- *VisualDSP++ 4.0 Product Release Bulletin*
- *VisualDSP++ Kernel (VDK) User's Guide*
- *Quick Installation Reference Card*

For hardware information, refer to your processors's hardware reference, instruction set reference (or programming reference), and data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

```
http://www.analog.com/processors/resources/technicalLibrary
```

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format

File	Description
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files requires a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the .PDF files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM) are located in the VisualDSP++ software installation's Help folder, and .PDF files are located in the Docs folder of your VisualDSP++ installation CD-ROM. The Docs folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.
- Double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.

Accessing Documentation From the Web

Download manuals at the following Web site:
`http://www.analog.com/processors/resources/technicalLibrary/manuals`

Select a processor family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Tools Manuals

To purchase EZ-KIT Lite™ and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions in this manual are identified and described as follows.

Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION TO VISUALDSP++

This manual describes VisualDSP++, a flexible management system that provides a suite of tools for developing processor applications and projects.

VisualDSP++ includes:

- Integrated Development and Debugging Environment (IDDE) with VisualDSP++ Kernel (VDK) integration
- C/C++ optimizing compiler with run-time library
- Assembler and linker
- Simulator software
- Example programs

This chapter contains the following topics:

- "VisualDSP++ Features" on page 1-2
- "License Management" on page 1-9
- "Project Development" on page 1-14
- "Code Development Tools" on page 1-27
- "Processor Projects" on page 1-44
- "VisualDSP++ Help System" on page 1-59

VisualDSP++ Features

VisualDSP++ includes all the tools needed to build and manage processor projects.

Integrated Development and Debugging

The VisualDSP++ IDDE provides complete graphical control of the edit, build, and debug process. In this integrated environment, you can move easily between editing, building, and debugging activities.

Code Development Tools

Depending on the code development tools purchased, VisualDSP++ includes one or more of the following components.

- C/C++ compiler with run-time library
- Assembler, linker, preprocessor, and archiver
- Loader and splitter
- Simulator
- EZ-KIT Lite™ evaluation system (must be purchased separately)
- Emulator (must be purchased separately)

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.

If your system is configured with third-party development tools, you can select the compiler, assembler, linker, or loader to use for a particular target build.

Source File Editing Features

VisualDSP++ simplifies tasks involving source files. All the activities necessary to create, view, print, move within, and locate information are easy to perform.

- **Edit text files.** Create and modify source files and view listing or map files generated by the code development tools.

  Source files are the C/C++ language or assembly language files that make up your project.

  Processor projects can include additional files such as data files and a Linker Description File (`.LDF`), which contains command input for the linker. For more information about `.LDF` files, see "Linker" on page 1-31.

- **Editor windows.** Open multiple editor windows to view and edit related files, or open multiple editor windows for a single file. The VisualDSP++ editor is an integrated code-writing tool that enables you to focus on code development.

- **Specify syntax coloring.** Configure options that specify the color of text objects viewed in an editor window.

  This feature enhances the view and helps locate portions of the text, because keywords, quotes, and comments appear in distinct colors.

- **Context-sensitive expression evaluation.** Move the mouse pointer over a variable that is in the scope to view the variable's value.

- **Status icons.** View icons that indicate breakpoints, bookmarks, and the current PC position.

- **View error details and offending code.** From the **Output** window's **Build** view, display error details by highlighting the error code (such as `cc0251`) and pressing the **F1** key. Double-click an error line to jump to the offending code in an editor window.

Project Management Features

VisualDSP++ provides flexible project management for the development of processor applications, including access to all the activities necessary to create, define, and build processor projects.

- **Define and manage projects.** Identify files that the code development tools process to build your project. Create this project definition once, or modify it to meet changing development needs.
- **Access and manage code development tools.** Configure options to specify how the code development tools process inputs and generate outputs. Tool settings correspond to command-line switches for code development tools. Define these options once, or modify them to meet your needs.
- **View and respond to project build results.** View project status while a build progresses and, if necessary, halt the build.

  Double-click on an error message in the **Output** window to view the source code causing the error, or iterate through error messages.
- **Manage source files.** Manage source files and track file dependencies in your project from the **Project** window to provide a display of software file relationships. VisualDSP++ uses code development tools to process your project and to produce a processor program. It also provides a source code control (SCC) interface, which enables you to access SCC applications without leaving the IDDE.

Debugging Features

While debugging your project, you can:

- **View and debug mixed C/C++ and assembly code.** View C/C++ source code interspersed with assembly code. Line number and symbol information help you to source-level debug assembly files.
- **Run command-line scripts.** Use scripts to customize key debugging features.
- **Use memory expressions.** Use expressions that refer to memory.
- **Use breakpoints to view registers and memory.** Quickly add and remove, and enable and disable breakpoints.
- **Set simulated watchpoints.** Set watchpoints on stacks, registers, memory, or symbols to halt program execution.
- **Statistically profile the target processor's PC** (JTAG emulator debug targets only). Take random samples and display them graphically to see where the program uses most of its time.
- **Linearly profile the target processor's PC** (Simulation only). Sample every executed PC and provide an accurate and complete graphical display of what was executed in your program.
- **Generate interrupts using streaming I/O.** Set up serial port (SPORT) or memory-mapped I/O.
- **Create customized register windows.** Configure a custom register window to display a specified set of registers.
- **Plot values from processor memory.** Choose from multiple plot styles, data processing options, and presentation options.

- **Trace program execution history.** Trace how your program arrives at a certain point and show reads, writes, and symbolic names.
- **View pipeline depth of assembly instructions**. Display the pipeline stage by querying the target processor(s) through the pipeline interface.

For details, refer to the *VisualDSP++ 4.0 Getting Started Guide* and VisualDSP++ online Help.

VDK Features

The VisualDSP++ Kernel (VDK) is a scalable software executive specially developed for effective operations on Analog Devices processors. The VDK is tightly integrated with VisualDSP++.

The kernel enables you to abstract the details of the hardware implementation from the software design. As a result, you can concentrate on the processing algorithms.

The kernel provides all the basic building blocks required for application development. Properties of the kernel can be characterized as follows.

- **Automatic**. VisualDSP++ automatically generates source code framework for each user-requested object in the user-specified language.
- **Deterministic**. VisualDSP++ specifies whether the execution time of a VDK API is deterministic.
- **Multitasking.** Kernel tasks (threads) are independent of one another. Each thread has its own stack.
- **Modular**. The kernel comprises various components. Future releases may offer additional functionality.

- **Portable**. Most of the kernel components can be written in ANSI Standard C or C++ and are portable to other Analog Devices processors.
- **Pre-emptive**. The kernel's priority-based scheduler enables the highest-priority thread not waiting for a signal to be run at any time.
- **Prototypical**. The kernel and VisualDSP++ create an initial file set based on a series of template files. The entire application is prototyped and ready to be tested.
- **Reliable**. The kernel provides run-time error checking.
- **Scalable**. If a project does not include a kernel feature, the support code is not included in the target system.

VisualDSP++ 4.0 Features

VisualDSP++ 4.0 includes the following new features and enhancements.

- **New processor support**. Refer to the processors listed in this book's Preface.
- **Compiled Simulation**. (Blackfin processors only)
  You no longer need to build an `.EXE` file from a `.DXE` file.
- **Project Wizard**. A new "wizard" has been added. The Project Wizard (**File** -> **New** -> **Project**) simplifies the process of creating a new project. For Blackfin processors, the Project Wizard also includes a Startup Code Wizard, offering configurable options such as cache, run-time initialization, and profiling.
- **Image Viewer**. The Image Viewer window includes more tools. Automatic refresh capability is now available.

- **Build Enhancements**. Now you can build multiple/selected files. You can build a batch of targets at once. You can build multiple targets called batch builds. When enabled, the **Output** window's **Build** view displays only the "names" of the files being built. You can save project group information to a `.DPG` file so you can restore that project group and share it conveniently. You can automatically execute a set of pre-build commands before building a project.
- **Help System Enhancements**. The VisualDSP++ 4.0 Help system now includes links to examples and demo programs. Grouped by processor families, the programs are further subdivided into the following categories: core and peripherals, audio effects, signal processing, video and graphics, VDK, and automation and scripting.
- **VisualDSP++ Configurator Enhancements**. Now you access the VisualDSP++ Configurator via the **New Session** dialog box (**Session** -> **New Session**). From the VisualDSP++ Configurator, test an emulator using the ICE Test utility.
- **Project Options Dialog Box Enhancements**. The **Project Options** dialog box no longer has tabbed pages. Instead, "pages" of options are accessible via a tree control on the left side of the dialog box. New pages are also available, such as **Pre-build**. Some of the tabbed pages available in VisualDSP++ 3.5 were rearranged (or split into multiple pages) to improve how options are grouped. Depending on the processor, additional pages (such as "**Workarounds**") may appear.

Consult VisualDSP++ online Help for procedures relating to these new features.

License Management

VisualDSP++ is a licensed software product. This section describes licensing options, license status, license installation, software registration, validation codes, product updates, and product serial numbers.

VisualDSP++ and EZ-KIT Lite evaluation systems are licensed products. They do not run unless a software license is installed. Three types of licenses are available: TST (evaluation), KIT (evaluation), and ADI (permanent).

- A “TST” (test drive) license provides unlimited (unrestricted) access to VisualDSP++ for emulation and simulation. You must register the software to receive a “TST” serial number, which expires 90 days after installation. After the 90 days, the software is inoperable.
- A “KIT” license grants 10 (ten) days to register and validate the installation. Once validated, the KIT licence extends to 90 (ninety) days of a full evaluation in total. At the end of the evaluation period, unless you have upgraded the KIT license to permanent, simulator and emulator connections become prohibited and the size of the user program is limited.
- An “ADI” license grants 30 (thirty) days to register and validate the installation for a permanent use. If you fail to register and validate your installation during the 30-day evaluation period, your copy of VisualDSP++ 4.0 becomes inoperable.

“KIT” serial numbers impose restrictions on VisualDSP++. These limitations do not prevent processor evaluation on the EZ-KIT Lite evaluation board, but encourage the purchase of a full (unrestricted) VisualDSP++ license.

Licensing Options

Two licensing options are available: *single-user* and *client*. A *server* license is required before you can install a client license (see Table 1-1).

Table 1-1. VisualDSP++ Licenses

License	Description
Single-User	Also called *node-locked* or *per-user* licenses, single-user licenses are locked to the machine ID of the host computer. Once installed, tools run only on that one machine. You may install and register the software up to three times (for example, at work, at home, and on a laptop computer). Use, however, is restricted to one installed software at a time.
Client	Client licenses are a client/server-based application. The server manages a pool of licenses installed on the server. One license is installed on the server for each purchased copy of VisualDSP++. In this model, you can have as many client installations as desired. When a client starts the software, it checks out a license from the server. When the software exits, the license is returned to the server. As long as licenses are available on the server, clients can access VisualDSP++. Example: Assume a license server has been set up with 10 licenses, and 20 client machines are installed in three labs. Ten simultaneous developers (any combination) can use the software. When the 11th client tries to use VisualDSP++, a message appears, stating that no more licenses are available. This allows sharing of the software resources in an environment that needs more locations than developers.
Server	Allows multiple users to access VisualDSP++ on computers sharing client licenses across a network. A server license must be installed prior to installing client licenses.

Server-based *floating licenses* consist of two parts: server and client. The server manages the license pool that is stored on the server. The clients "check out" licenses when the software is started and return licenses to the server when developers exit VisualDSP++.

License Status

The **Licenses** page of the **About VisualDSP++** dialog box displays the status of all recognized licenses.

For complete details about licenses, refer to VisualDSP++ online Help.

Temporary Licenses

A temporary license indicates the number of days remaining before the product can no longer be used. Test drive versions of VisualDSP++ (serial number beginning with "`TST`") carry temporary licenses.

An unrestricted version of VisualDSP++ includes its permanent license. If you do not install the validation code after purchasing a full (unrestricted) license, the status of the license is marked "`Not Validated (Expiring in X days)`". Install the validation code to change the status to "Permanent."

Valid Versus Expired Licenses

An expired license is indicated by "`Expired`". A valid license is indicated by "`Permanent`" unless it is for an EZ-KIT Lite evaluation system. A valid license for an EZ-KIT Lite is indicated by "`Permanent, Restricted`".

Client Licenses

When a client license is installed, the "`server_name`" appears under **Serial Number**, "`client`" appears under **Family**, and "`use_server`" appears under **Status**.

License Installation

After installing VisualDSP++, you must license the software. Licensing involves these three tasks:

1. Installing a single-user or client serial number

   Note that a server license must be installed before client licenses can be installed.

2. Registering product(s)

3. Entering validation code(s)

Perform license management activities (except server licensing) within VisualDSP++ by using the **About VisualDSP++** dialog box. See the online Help for detailed installation, registration, and validation procedures.

Test drives require online registration to receive a "TST" serial number, which expires 90 days after installation. Test drives do not require a validation code.

Installing a License Shipped With an EZ-KIT Lite Evaluation System

Versions of VisualDSP++ used on EZ-KIT Lite evaluation systems require online registration and a validation code. The EZ-KIT Lite evaluation system's "KIT" serial number is located on the label attached to the back of the CD wallet.

"KIT" serial numbers impose these restrictions on VisualDSP++:

- The size of a user program is limited.
- No connections to simulator or emulator sessions are allowed.

- Only one EZ-KIT Lite board can be connected to the host PC and debugged at a time.
- The EZ-KIT Lite hardware must be connected and powered up to use VisualDSP++ with a "KIT" license.

These limitations do not prevent processor evaluation on the EZ-KIT Lite evaluation board, but they encourage the purchase of a full (unrestricted) VisualDSP++ license.

VisualDSP++ Product Upgrades

From time to time, Analog Devices releases new software versions.

The upgrade procedure does not change the previous version's folder structure or license file. The new installation process uses the previous version's path and license.

Check the Analog Devices Web site to ensure that you have the latest software version.

Product Serial Numbers

Product serial numbers are located on product CD sleeves. A product's serial number can also be viewed from within VisualDSP++.

If you cannot locate a serial number, contact your local sales representative or Analog Devices sales by:

- Sending e-mail to: dsptools.support@analog.com
- Phoning 1-800-ANALOGD (1-800-262-5643)

Provide details about the exact products, versions, and operating system being used.

Within VisualDSP++, view product serial numbers from the **Licenses** page of the **About VisualDSP++** dialog box, accessible from the **Help** menu.

Project Development

During project development, VisualDSP++ helps you interactively observe and alter the data in the processor and in memory.

Overview of Programming With VisualDSP++

Programming effectively with VisualDSP++ depends on how well you master a four-step process. You must learn how to:

1. Work with VisualDSP++
2. Implement structured software design with VisualDSP++
3. Optimize performance with VisualDSP++
4. Test and debug your programs with VisualDSP++

Working With VisualDSP++:
You should have a working knowledge of VisualDSP++, the front end for all available targets and platforms. You should know how and when to use its various features and have a firm foundation in these project basics:

- Working with "property pages". These pages of the **Project Options** dialog box provide options analogous to command-line switches.
- Setting up debug sessions. Know the distinctions between the three development stages: simulation, evaluation (via an EZ-KIT Lite evaluation system), and emulation.
- Understanding how program sections and memory segments relate to physical processor memory. Become familiar with Expert Linker.
- Accessing peripherals. This task includes setting up and handling interrupts in both C and assembly.

Designing Structured Software With VisualDSP++:
Consider elements of software design, code reuse, and interoperability. If you are new to embedded systems, try to acquire a clear understanding of:

- The role of and motivation behind component software
- The role of an RTOS
- How to use VDK to manage multiple threads of execution and the communication between those threads

Optimizing Performance With VisualDSP++:
At this stage, you should understand how to access the features of the processor and how to use a structured approach to develop software. Next, optimize your software to take full advantage of the processor's computational power. This entails:

- Understanding the compiler optimizer
- Writing mixed C and assembly programs
- Accessing C/C++ data structures in assembly
- Harnessing the power of C++
- Setting up and using overlays
- Configuring emulation L1 memory for cache versus SRAM with cache visualization
- Using statistical profiling

Testing and Debugging With VisualDSP++:
At this stage, you should have a good understanding of the various facilities available for producing optimal software. The last step, applying software testing and debugging techniques, includes:

- Collecting and viewing data using the advanced plot windows
- Using compiled simulation
- Using ActiveX and COM Automation to create regression test environments and taking advantage of interoperability with other applications

Project Development Stages

The typical project includes three phases: *simulation*, *evaluation*, and *emulation*. These phases are shown in Figure 1-1.

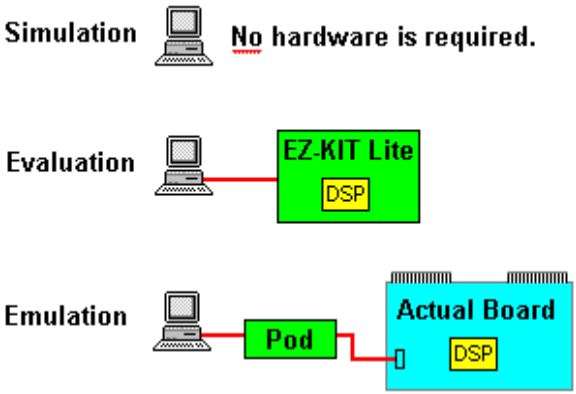

Figure 1-1. Project Development Stages

VisualDSP++ provides tools for use during each of these phases.

Simulation

Project development typically begins in a simulation environment while hardware engineers are developing the new hardware (cell phone, computer, and so on). Simulation mimics system memory and I/O, which enables portions of the target system hardware to be viewed. A *simulator* is software that mimics the behavior of a processor. Running VisualDSP++ with a simulation target (without a physical processor) enables you to build, edit, and debug your program, even before a processor is manufactured.

Evaluation

Use an EZ-KIT Lite evaluation system in your project's early planning stage to determine the processor that best fits your needs. Your PC connects to the EZ-KIT Lite board via a cable, enabling you to monitor processor behavior.

Emulation

Once the hardware is ready, move directly to a JTAG *emulator*, which connects your PC to the actual processor target board. Emulators provide fast communications between the board and your PC. An emulator enables application software to be downloaded and debugged from within VisualDSP++. Emulator software performs the communications that enable you to see how your code affects processor performance.

Targets

A *target* (or debug target) refers to the communication channel between VisualDSP++ and a processor (or group of processors). A target can be a simulator, EZ-KIT Lite evaluation board, or an emulator. Your system can include multiple targets.

For example, the JTAG emulator communicates with one or more physical devices over the host PC's PCI bus, and the Apex-ICE™ emulator communicates with a device via the PC's USB port.

Simulation Targets

A *simulation target*, such as the ADSP-2106x Family Simulator, is a pure software module and does **not** require the presence of a processor for debugging.

During simulation, VisualDSP++ reads an executable (`.DXE`) file and executes it in software, similar to the way a processor executes a processor image in hardware. VisualDSP++ simulates the memory and I/O devices specified in an `.LDF` file.

EZ-KIT Lite Targets

An *EZ-KIT Lite target* is a development board used to evaluate a particular processor. Analog Devices provides several EZ-KIT Lite evaluation systems, which include demonstration programs.

Emulation Targets

An *emulation target* is a module that controls a physical processor connected to a JTAG emulator system. For example, the Summit-ICE™ emulator communicates with one or more physical devices through the host PC's PCI bus, and the Apex-ICE emulator communicates with a device through the PC's USB port.

Platforms

A *platform* refers to the configuration of processors with which a target communicates. Several platforms may exist for a given debug target. For example, if three emulators are installed on your system, you might select emulator 2 as the platform that you want to use. The platform that you use depends on your project development stage. (See Table 1-2.)

Table 1-2. Development Stages and Supported Platforms

Stage	Platform
Simulation	Typically one or more processors of the same type. By default, the platform name is the identical simulator.
Evaluation	An EZ-KIT Lite evaluation system
Emulation	Any combination of devices. You must configure the platform for a particular target with the VisualDSP++ Configurator. When the debug target is a JTAG emulator, platform refers to a JTAG chain.

Hardware Simulation

When connected to a simulation target in VisualDSP++, you can simulate the following hardware conditions.

- Random interrupts that can occur during program execution
- Data transfer through the processor's I/O pins
- Processor booting from a PROM or host processor

Setting up VisualDSP++ to generate random interrupts during program execution enables you to exercise interrupt service routines (ISR) in your code.

Debugging Overview

Once you have successfully built a processor project and generated an executable file, you can debug the project. Projects developed in VisualDSP++ are run as hardware and software *debug sessions*.

In Table 1-3, check marks (✔) indicate which debugging tools are available during the process of building and debugging a program.

Table 1-3. Tools Available During Simulation, Evaluation, and Emulation

Tool	Simulation	Evaluation	Emulation
Linear profiles on page 3-8	✔		
Interrupts on page 3-16	✔		
Streams on page 3-16	✔		
Traces (SHARC processors only) on page 3-9	✔		

Table 1-3. Tools Available During Simulation, Evaluation, and Emulation

Tool	Simulation	Evaluation	Emulation
Pipeline Viewer (not SHARC processors) on page C-2	✔		
Breakpoints on page 3-13	✔	✔	✔
Watchpoints on page 3-14	✔		
Hardware breakpoints on page 3-14			✔
Plotting on page 3-19	✔	✔	✔
Statistical profiles on page 3-8			✔

You can attach to and control the operation of any Analog Devices processors or simulator. Download your application code to the processor and use VisualDSP++'s debugging facilities to ensure that your application functions as desired.

VisualDSP++ is a window into the inner workings of the target processor or simulator. From this user interface, you can:

- Run, step, and halt the program and set breakpoints and watchpoints
- View the state of the processor's memory, registers, and stacks
- Perform a cycle-accurate statistical profile or linear profile
- Perform integrated multiprocessor debugging (emulator sessions only)

VisualDSP++ Kernel

A project can optionally include the VisualDSP++ Kernel (VDK), which is a software executive between algorithms, peripherals, and control logic.

The **Project** window's **Kernel** tab accesses a tree control for structuring and scaling application development. From this tree control, you can add, modify, and delete Kernel elements such as thread types, boot threads, round-robin priorities, semaphores, events, event bits, interrupts, and device drivers.

Two VDK-specific windows, **VDK State History** and **Target Load**, provide views of VDK information. Another VDK window, **VDK Status**, provides thread status data when a VDK-enabled program is halted. Refer to the *VisualDSP++ 4.0 Kernel (VDK) User's Guide* for complete details.

Program Development Steps

In the VisualDSP++ environment, program development consists of the following steps.

1. Create a project.
2. Configure project options.
3. Add and edit project source files.
4. Specify project build options.
5. Build a debug version (executable file) of the project.
6. Create a debug session and load the executable.
7. Run and debug the program.
8. Build a release version of the project.

By following these steps, you can build projects consistently and accurately with minimal project management. This process reduces development time and lets you concentrate on code development.

These steps, described below, are covered in detail in the online Help and in the "Basic Tutorial" chapter of the *VisualDSP++ 4.0 Getting Started Guide*.

Step 1: Create a Project

All development in VisualDSP++ occurs within a project. The project (`.DPJ`) file stores your program's build information: source files list and development tools option settings.

VisualDSP++ includes a Project wizard that simplifies the creation of a new project. Refer to the *VisualDSP++ Getting Started Guide* for a tutorial or to VisualDSP++ online Help.

Step 2: Configure Project Options

Define the target processor and set up your project options (or accept default settings) before adding files to the project. The **Project Options** dialog box provides access to project options (also called property pages), which enable the corresponding build tools to process the project's files correctly.

Step 3: Add and Edit Project Source Files

A project normally contains one or more C, C++, or assembly language source files. After creating a project and defining its target processor, add new or existing files to the project by importing or writing them. Use the VisualDSP++ editor to create new files or edit any existing text files.

Adding Files to Your Project

You can add any type of file to the project. The development tools selectively process only recognized file types when you build the project.

Creating Files to Add to Your Project

You can create new text files. The editor can read or write text files with arbitrary names. Adding files to your project updates the project's file tree in the **Project** window.

Editing Files

You can edit the file(s) that you add to the project. To open a file for editing, double-click on the file icon in the **Project** window.

The editor has a standard Windows-style user interface and supports normal editing operations and multiple open windows. You can customize language- and processor-specific syntax coloring, and create and search for bookmarks.

Managing Project Dependencies

Project dependencies control how source files use information in other files, and consequently determine the build order. VisualDSP++ maintains a makefile, which stores dependency information for each file in the project. VisualDSP++ updates dependency information when you change the project's build options, add a file to the project, or choose **Update Dependencies** from the **Project** menu.

Step 4: Specifying Project Build Options

After creating a project, setting the target processor, and adding or editing the project's source files, configure your project's build options. Specify options or accept the default options in VisualDSP++ before using the development tools that create your executable file. You can specify options for a whole project or for individual files, or you can specify a custom build.

VisualDSP++ retains your changes to the build options. Settings reflect your last changes, not necessarily the original defaults.

Configuration

A project's *configuration* setting controls its build. By default, the choices are **Debug** or **Release.**

- Selecting **Debug** and leaving all other options at their default settings builds a project that can be debugged. The compiler generates debug information.
- Selecting **Release** and leaving all other options at their default settings builds a project with limited or no debug capabilities. Release builds are usually optimized for performance. Your test suite should verify that the Release build operates correctly without introducing significant bugs.

You can modify VisualDSP++'s default operation for either configuration by changing the appropriate entries on the **Compile**, **Assemble**, and **Link** pages. You can create custom configurations that include the build options and source files that you want.

Project-Wide File and Tool Options

Next, you must decide whether to use project-wide option settings or individual file settings.

For projects built entirely within VisualDSP++ with no pre-existing object or archive (library) files, you typically use project-wide options. New files added to the project inherit these settings.

Individual File and Tool Options

Occasionally, you may want to specify tool settings for individual files. Each file is associated with two property pages: a **General** page, which lets you choose output directories for intermediate and output files, and a tool-specific property page (**Compile**, **Assemble**, **Link**, and so on), which lets you choose options. For information about each tool's options, see the online Help or the manual for each tool.

Step 5: Build a Debug Version of the Project

Next, build a debug version of the project.

Status messages from each code development tool appear in the **Output** window as the build progresses.

The output file type *must* be an executable (`.DXE`) file to produce debugger-compatible output.

Step 6: Create a Debug Session and Load the Executable

After successfully building an executable file, set up a *debug session*. You run projects that you develop as hardware or software sessions. After specifying target and processor information, load your project's executable file. On the **General** page in the **Preferences** dialog box, you can configure VisualDSP++ to load the file automatically and advance to the `main` function of your code.

Step 7: Run and Debug the Program

After successfully creating a debug session and building and loading your executable program, run and debug the program.

If the project is not current (has outdated source files or dependency information), VisualDSP++ prompts you to build the project before loading and debugging the executable file.

Step 8: Build a Release Version of the Project

After you finish debugging your application, build a Release version of your project to run on the product's processor.

Code Development Tools

This section describes the following development tools.

- C/C++ compiler with run-time libraries
- Assembler
- Linker
- Expert Linker
- Preprocessor
- Archiver
- Splitter
- Loader

Available code development tools differ, depending on the processor. The options available on the pages of the **Project Options** dialog box enable you to specify tool preference.

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format, Debugging Information Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.

If your system is configured with third-party development tools, you can select the compiler, assembler, or linker to be used for a particular target build.

Compiler

The compiler processes C/C++ programs into assembly code. The term *compiler* refers to the compiler utility shipped with the VisualDSP++ software.

The compiler generates a linkable object file by compiling one or more C/C++ source files. The compiler's primary output is a linkable object file with a `.DOJ` extension.

To specify compiler options for your build, choose **Project** -> **Project Options**. From the ensuing **Project Options** dialog box, expand **Compile** and click a subpage.

Compiler options are grouped into the categories described in Table 1-4.

Table 1-4. Groups of Compiler Options

Category	Provides
General	Optimization, compilation, and termination options
Preprocessor	Macro and directory search options
Processor	Processor-specific options
Warning	Warning and error reporting options
Workarounds	Workarounds for specific hardware errata

The available compile pages and options depend on your target processor and your code development tools.

For complete information about compile options, refer to your processor's VisualDSP++ 4.0 C/C++ Compiler and Library manual and VisualDSP++ online Help.

C++ Run-Time Libraries

You must be running VisualDSP++ to use the C++ run-time libraries.

The C and C++ run-time libraries are collections of functions, macros, and class templates that can be called from source programs. Many functions are implemented in the processor assembly language.

C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming languages. These operations include memory allocations, character and string conversions, and math calculations. The libraries also include multiple signal processing functions that ease processor code development. The run-time library simplifies software development by providing code for a variety of common needs.

The compiler provides a broad collection of C functions including those required by the ANSI standard and additional Analog Devices-supplied functions of value for processor programming. This release of the compiler software includes both the Standard C Library and the Abridged Library, a conforming subset of the Standard C++ Library.

For more information about the algorithms on which many of the C library's math functions are based, refer to the Cody and Waite text *Software Manual for the Elementary Functions* from Prentice Hall (1980).

For more information about the C++ library portion of the ANSI/ ISO Standard for C++, refer to the Plauger text *Draft Standard C++ Library* from Prentice Hall (1994) (ISBN: 0131170031).

Assembler

The assembler generates an object file by assembling source, header, and data files. The assembler's primary output is an object file with a `.DOJ` extension.

To specify assembler options, choose **Project** -> **Project Options**, and click **Assemble** (in the **Project Options** dialog box).

Assembler terms are defined as follows.

Instruction set

Set of assembly instructions that pertain to a specific processor. For information about the instruction set, refer to your processor's hardware documentation.

Preprocessor commands

Commands that direct the preprocessor to include files, perform macro substitutions, and control conditional assembly

Assembler directives

Directives that tell the assembler how to process source code and set up processor features. Use directives to structure your program into logical *segments* or sections that support the use of a Linker Description File (`.LDF`) to construct an image suited to the target system.

For detailed information, refer to the *VisualDSP++ 4.0 Assembler and Preprocessor Manual* or VisualDSP++ online Help

Linker

The linker links separately assembled files (object files and library files) to produce executable (.DXE) files, shared memory (.SM) files, and overlay (.OVL) files, which can be loaded onto the target.

The linker's output files (.DXE, .SM, .OVL) are binary, executable, and linkable files (ELF). To make an executable file, the linker processes data from a Linker Description File (.LDF) and one or more object (.DOJ) files. The executable files contain program code and debugging information. The linker fully resolves addresses in executable files.

To specify linker options, choose **Project** -> **Project Options**, and click **Link** tab (on the **Project Options** dialog box). From the **Link** page, select a **Category** of options. Linker options are grouped into the following categories.

- General
- LDF Preprocessing
- Elimination
- Processor

Linker terms are defined as follows.

Link against

Functionality that enables the linker to resolve symbols to which multiple executables refer. For instance, shared memory (.SM) executable files contain sections of code that other processor executable (.DXE) files link against. Through this process, a shared item is available to multiple executable files without being duplicated.

Link objects

Object files (.DOJ) that become linked and other items, such as executable (.DXE, .SM, .OVL) files, that are linked against

.LDF file

File that contains the commands, macros, and expressions that control how the linker arranges your program in memory

Memory

Definitions that provide a description of your target processor system to the linker

Overlays

Files that your overlay manager swaps in and out of run-time memory, depending on code operations. The linker produces overlay (.OVL) files.

Sections

Declarations that identify the content for each executable file that the linker produces

For detailed information, refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* or VisualDSP++ online Help

A Linker Description File (.LDF) describes the target system and maps your program code with the system memory and processors.

The .LDF file creates an executable file by using:

- The target system memory map
- Defined segments in your source files

The parts of an .LDF file, from the beginning to the end of the file, are described as follows.

- Memory map – describes the processor's physical memory (located at the beginning of the .LDF file)
- SEARCH_DIR, $LIBRARIES, and $OBJECTS commands – define the path names that the linker uses to search and resolve references in the input files
- MEMORY command – defines the system's physical memory and assigns labels to logical segments within it. These logical segments define program, memory, and stack memory types.
- SECTIONS command – defines the placement of code in physical memory by mapping the sections specified in program files to the sections declared in the MEMORY command. The INPUT_SECTIONS statement specifies the object file that the linker uses to resolve the mapping.

For details, refer to the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Expert Linker

Expert Linker is a graphical tool that enables you to:

- Define a target processor's memory map
- Place a project's object sections into that memory map
- View how much stack or heap has been used after you run a processor program

This interactive tool speeds up the configuration of system memory. It uses your application's target memory description, object files, and libraries to create a memory map that you can manipulate to optimize your system's use of memory.

Expert Linker works with the linker. For more information about linking, refer to the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Expert Linker graphically displays the available project information in an `.LDF` file as input. This information includes object files, LDF macros, libraries, and target memory descriptions. Use the drag-and-drop function to arrange the object files in a graphical memory-mapping representation. When you are satisfied with the memory layout, generate the executable file (`.DXE`) via VisualDSP++ project options.

You can use default `.LDF` files that come with VisualDSP++ or the Expert Linker wizard to create and customize a new `.LDF` file.

When opened in a project that already includes an `.LDF` file, Expert Linker parses the `.LDF` file and graphically displays the target processor's memory map and the object mappings. The memory map appears in the **Expert Linker** window (Figure 1-2 on page 1-36). Use this display to modify the memory map or the object mappings. When the project is ready to be built, Expert Linker saves the changes to the `.LDF` file.

Expert Linker can graphically display space allocated to program heap and stack. After you load and run your program, Expert Linker indicates the used portion of the heap and stack. You can then reduce the size of the heap or stack to minimize the memory allocated for the heap and stack. Freeing up memory in this way enables it to be used for storing other things like processor code or data.

You can launch the Expert Linker (see Figure 1-2) from VisualDSP++ in three ways:

- Double-click the `.LDF` file in the **Project** window.
- Right-click the `.LDF` file in the **Project** window to display a menu and then choose **Open in Expert Linker**.
- From the VisualDSP++ main menu, choose **Tools**, **Expert Linker**, and **Create LDF**.

Expert Linker Window

The Expert Linker window (Figure 1-2) enables you to modify the memory map or the object mappings. You can specify a color for each type of object (internal memory, external memory, unused memory, reserved memory, output sections, object sections, overlays in live space, and overlays in run space). The objects are displayed in color when you view the **Memory Map** pane in graphical memory map mode. When the project is ready to be built, Expert Linker saves the changes to the `.LDF` file.

The **Expert Linker** window contains two main panes:

- The **Input Sections** pane displays a tree structure of the input sections.
- The **Memory Map** pane displays each memory map in a tree or graphical representation.

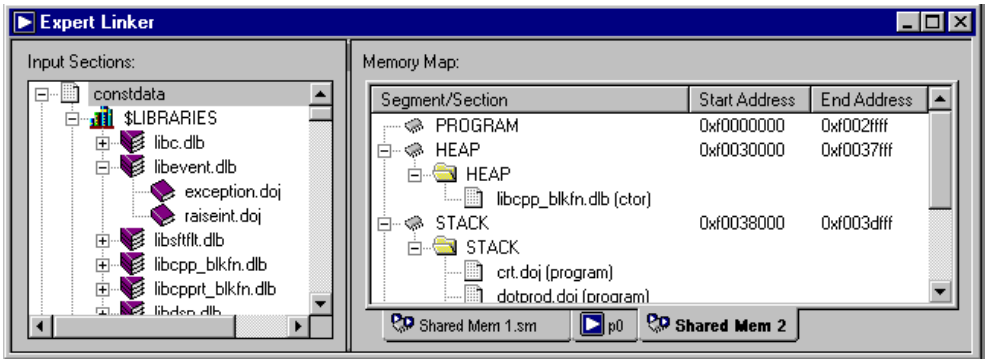

Figure 1-2. Expert Linker Window

You can dock or float the **Expert Linker** window in the VisualDSP++ main window.

Memory Map Pane Right-Click Menu

Table 1-5 describes the commands on the Memory Map right-click menu.

Table 1-5. Memory Map Pane Right-Click Menu

Command	Purpose
View Mode→Memory Map Tree	Displays the memory map in tree mode
View Mode→Graphical Memory Map	Displays the memory map in graphical blocks
View→Mapping Strategy (Pre-Link)	Displays the memory map, which shows where you intended to place object sections
View→Link Results (Post-Link)	Displays the memory map, which shows where the object sections are actually placed
New→Memory Segment	Opens the **Memory Segment Properties** dialog box, from which you specify the name, address range, type, width, and so on of the memory segment that you want to add

Table 1-5. Memory Map Pane Right-Click Menu (Cont'd)

Command	Purpose
New→Output Section	Adds an output section to the selected memory segment **Note**: Right-click on a memory segment to access this command.
New→Shared Memory	Opens the **Shared Memory Properties** dialog box, from which you specify the name of the shared memory output file and processors that share the memory This command is not available on single-processor systems.
New→Overlay	Opens the **Overlay Properties** dialog box, from which you add a new overlay to the selected output section or memory segment **Note**: The new overlay's run space is in the selected output section.
Delete	Deletes the selected object
Pin-to-Output Section	Pins an object section to an output section to prevent it from overflowing to another output section This command is available only after you right-click on an object section that is part of an output section set to overflow to another section.
View Section Contents	Opens the **Section Contents** dialog box, which displays the contents of the input or output section This command is available only after you link or build the project and then right-click on an input or object section.

Table 1-5. Memory Map Pane Right-Click Menu (Cont'd)

Command	Purpose
Add Hardware Page Overlay Support	Sets up hardware overlay live and run spaces for all available hardware pages by: a) Checking if memory segments are currently defined in all hardware pages. If memory segments are located, you are queried about whether to delete those segments. b) Creating a memory segment containing an overlay (live space) in each hardware page c) Creating a memory segment containing all overlay run spaces in hardware page 0 d) Creating a default mapping for each overlay. The default mapping maps objects containing the section, "`pmpage0`" to the hardware overlay on PM page 0, "`pmpage1`" to PM page 1, "`dmpage0`" to DM page 0, and so on.
View Symbols	Opens the **View Symbols** dialog box and displays the symbols for the project, overlay, or input section This command is available after you link the project and then right-click on the **Memory Map** pane for a processor, memory segment, output section, or input section.
Expand All	Expands all items in the memory map tree to make their contents visible
View Legend	Opens the **Legend** dialog box, which shows all possible icons in the tree window, with a brief description of each icon. The **Colors** page displays a list of colors used in the graphical memory map. You can specify each object's color.
View Global Properties	Opens the **Global Properties** dialog box for the selected object. The dialog box's title and content depend on the selected object.

Stack and Heap Usage

Expert Linker enables you to adjust the size of the stack and heap, and make better use of memory.

Expert Linker can:

- Locate stacks and heaps and fill them with a marker value

  This operation occurs after you load the program into a processor target. The stacks and heaps are located by their memory segment names, which may vary across processor families.

- Search the heap and stack for the highest memory locations written to by the processor program

  This operation occurs when the target halts after you run the program. Assume that these values are the start of the unused portion of the stack or heap. Expert Linker updates the memory map to show how much of the stack and heap are unused.

Be aware of the following stack and heap restrictions.

- The heap, stack, and system stack must be defined in output sections named `HEAP`, `STACK`, and `SYSSTACK`, respectively.
- The heap, stack, and system stack must be the only items in those output sections. You cannot place other objects in those output sections.

For other processor families, the restrictions on memory segment names differ according to what is used in the default `.LDF` files. If you do not heed these restrictions, you cannot view stack and heap usage after running your program.

Figure 1-3 shows an example memory map after you run a SHARC program.

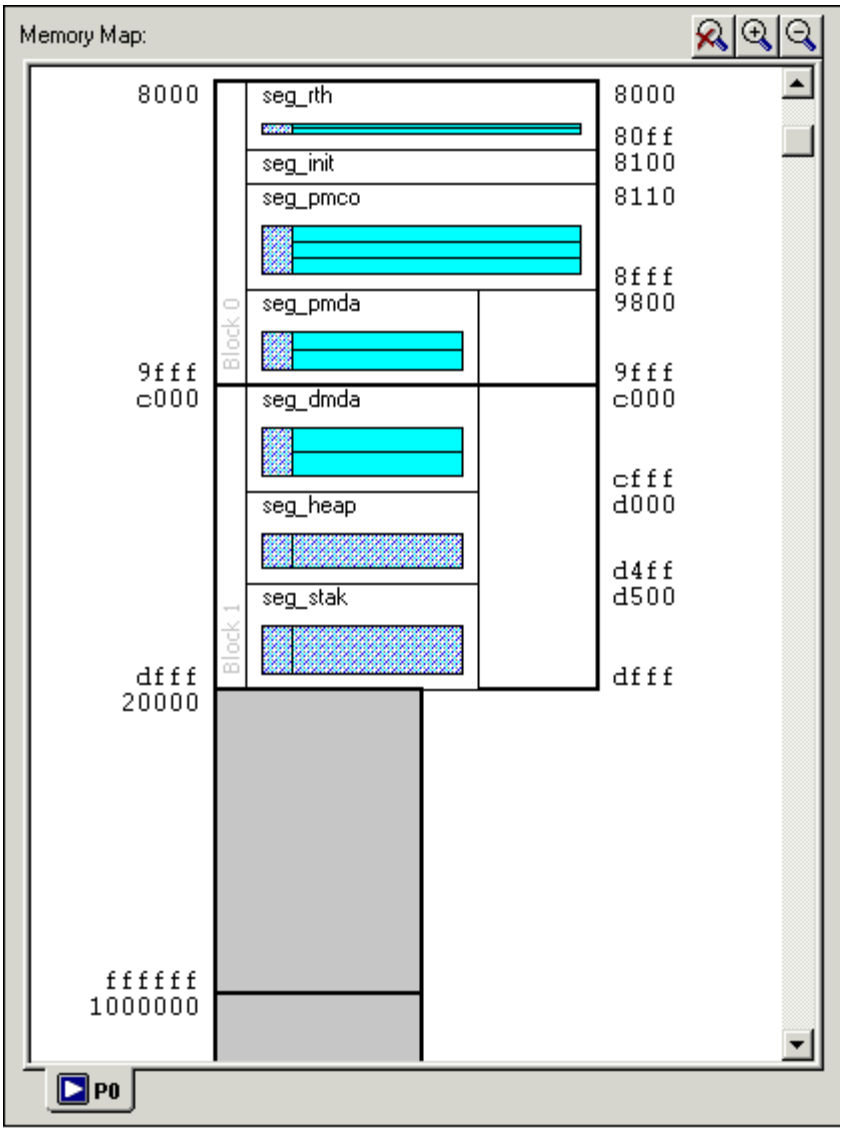

Figure 1-3. Memory Map Example After Running a SHARC Program

Archiver

The VisualDSP++ archiver (`elfar.exe`) combines object (`.DOJ`) files into library (`.DLB`) files, which serve as reusable resources for project development. The linker searches library files for routines (library members) referred to by other objects and links them in your executable program.

Run the archiver from within VisualDSP++ or from the command line. From VisualDSP++, create a library file as your project's output.

To modify or list the contents of a library file (or perform other operations on it), you must run the archiver from a command line. For details, refer to the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Splitter

The splitter (`elfspl21k.exe`) processes executable files to prepare non-bootable programmable read-only memory (PROM) image files. These files are executed from the processor's external memory.

The splitter's primary output is a PROM file with these extensions:

- `.S_#` (SHARC processors)
- `.H_#` (SHARC processors)
- `.STK` (SHARC processors)
- `.LDR` (Blackfin and TigerSHARC processors)

For TigerSHARC processors, output from the splitter is 32 bit. For SHARC processors, output from the splitter is 32 bit, 40 bit, 48 bit, or 64 bit.

To specify splitter options, choose **Project**, **Project Options**, and click the **Split** page.

Splitter terms are defined as follows.

Non-bootable PROM-image files

The splitter's output, which consists of PROM files that cannot be used to boot-load a system

Splitter

The splitter application, such as `elfspl21k.exe`, contained in the software release

For more information about the splitter and options used to generate loader files, refer to the *VisualDSP++ 4.0 Loader Manual* or VisualDSP++ online Help

Loader

The loader (`elfloader.exe`) generates boot-loadable files by processing executable files in addition to a loader kernel. The loader output (`.LDR`) file enables the processor to boot from an external device (host or ROM).

The loader creates programs that execute from internal memory. The splitter generates programs that execute from external memory.

To specify loader options, choose **Project**, **Project Options**, and open the **Load** pages.

Loader terms are defined as follows:

Boot kernel

The executable file that performs memory initialization on the target

Boot-loadable file

The loader's output (`.LDR`), which contains the boot loader and the formatted system configurations. This is a bootable image file.

Boot loading

The process of loading the boot loader, initializing system memory, and starting the application on the target

Loader

The loader application, such as `elfloader.exe`, contained in the software release

For more information about the loader, refer to the *VisualDSP++ 4.0 Loader Manual* or VisualDSP++ online Help.

Processor Projects

Your goal is to create a program that runs on a single-processor (or multiprocessor) system. A *project* is the structure where programs are built. All development in VisualDSP++ occurs within a project.

A project refers to the collection of source files and tool configurations used to create a processor program. The project file (`.DPJ`) stores program build information.

VisualDSP++ provides flexibility for setting up projects. You configure settings for code development tools and configurations, and you specify build settings for the project and for individual files. You can set up folders that contain your source files. A project can include VDK support.

Use the **Project** window to manage projects from start to finish. Within the context of a project, you can:

- Specify code development tools
- Specify project-wide and individual-file options for Debug or Release configurations of project builds
- Create source files

VisualDSP++ facilitates movement among editing, building, and debugging activities.

Project Options

Project options apply to the entire project. Specify project options in the **Project Options** dialog box. Figure 1-4 shows an example of this dialog box.

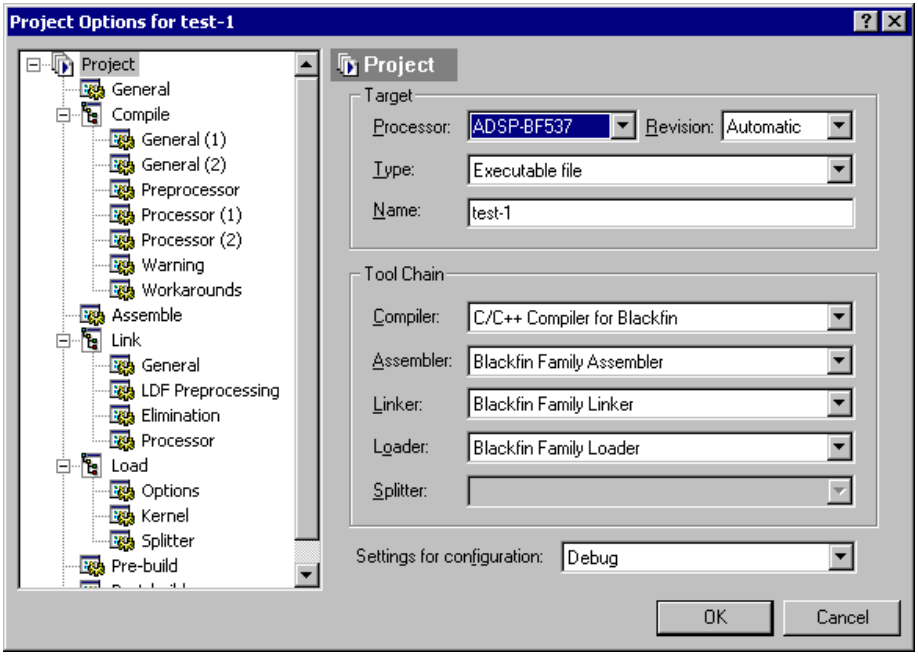

Figure 1-4. Project Options Dialog Box Showing Project Page

For each code development tool (compiler, assembler, linker, splitter, and loader), a one or more pages provide options that control how each tool processes inputs and generates outputs. The available pages depend on your target. Each option corresponds to an individual tool's command-line switch. You can define these options once or modify them to meet changing development needs.

Tools can also be accessed from the operating system's command line.

Project options also specify the following information.

- Project target
- Tool chain
- Output file directories
- Pre- and post-build options

Project Groups

A *project group* enables you to work with a number of projects at once. A project group can be empty or contain any number of projects. Opening a project adds it to the project group. Closing a project removes it from the project group. Similar functionality is found in Microsoft Visual Studio.

The **Project** window (Figure 1-5) displays the project group icon and the projects opened in that workspace.

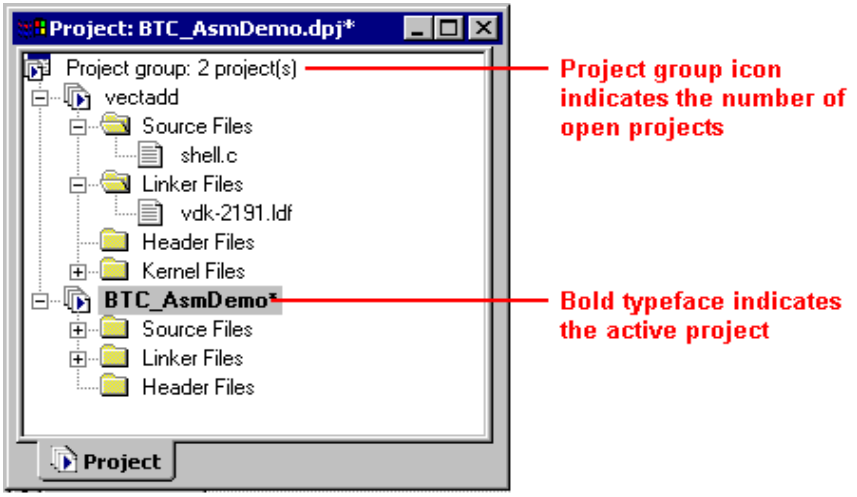

Figure 1-5. Project Window

Each workspace has one project group. When you switch among workspaces, the project group is loaded and the same set of projects are opened just as when you last closed the workspace.

One project is active at a time. The active project responds to commands and messages from menus and toolbars. The **Project** window displays the active project with bold typeface. A **Project** box, located by default with the toolbar buttons, displays the name of the active project (see Figure 1-6).

Figure 1-6. Project Box Showing the Active Project

Though commands are sent to the active project, they may also be carried out by a project on which the active project depends. For example, assume that project A is active and depends on project B. Executing a **Rebuild All** command on project A builds project B first. The same logic applies to the **Clean** command, which deletes intermediate and target files.

Exporting a makefile exports one makefile for each open project. In the makefile of a project depending on another project, one sub-target is created for each project on which it depends. Thus, building a project builds all dependent projects first.

Project Group Files

You can save project group information to a file so you can restore that project group and share it conveniently.

The project group file (.DPG), which is in XML format, contains a list of project entries. Each project entry corresponds to a project in the group and contains project information, including the path to the project file (.DPJ) and its dependent projects. Batch build specifications are saved in the .DPG file for later use (so you can load and execute them without re-specifying the same build targets). In the **Project** window, the root node shows the project group's file name without an extension.

Source Code Control (SCC)

VisualDSP++ includes Source Code Control (SCC), which enables you to use the Microsoft Common Source Code Control (MCSCC) interface to connect the VisualDSP++ IDDE to SCC applications installed on your machine.

Various SCC products (such as Microsoft Visual SourceSafe or PVCS Version Manager) support the MCSCC interface. From VisualDSP++ interface, you can access the commonly used features of these applications without leaving the IDDE. You can launch the SCC application from the plug-in menu to use non-supported features.

When you create a project, you are prompted to add the project to SCC. When you open a project in the IDDE, the SCC plug-in connects to the selected SCC application and locates a controlled copy of the project and its source files. If a controlled copy is not located, the SCC application must locate it. Typically, you are queried to browse for it. If the controlled copy is successfully found or added, the plug-in keeps its application-specific path in the project file and reconnects with this path in the future. You can subsequently reconnect to the controlled copy without having to browse for it.

Operations executed on large numbers of files tend to take longer to finish. A message box provides status information by displaying the operation currently executing. A button on the message box cancels the operation.

The **Output** window's **Console** view displays finished operations. Messages indicate what has been done. Warnings and error messages may also appear in the **Output** window.

SCC applications provide dialog boxes and GUI displays for some file operations such as show history, show difference, and show properties. These operations can be run from VisualDSP++.

For complete details, refer to VisualDSP++ Help.

Makefiles

Use a *makefile* (`.MAK` or `.MK`) to automate builds within VisualDSP++. The output make rule is compatible with the gnumake utility (GNU Make V3.77 or higher) or other make utilities. VisualDSP++ generates a project makefile that controls the orderly sequence of code generation in the target. You can also export a makefile for use outside of VisualDSP++. For more information about makefiles, go to:

```
http://www.gnu.org/manual/make/
```

A project can have multiple makefiles, but only one makefile can be enabled (active).

The project in Figure 1-7 includes an active makefile (indicated by).

The active makefile, with its explicit `gmake` command line, builds the project. When no makefile is enabled for a project, VisualDSP++ uses specifications configured in the **Project Options** dialog box.

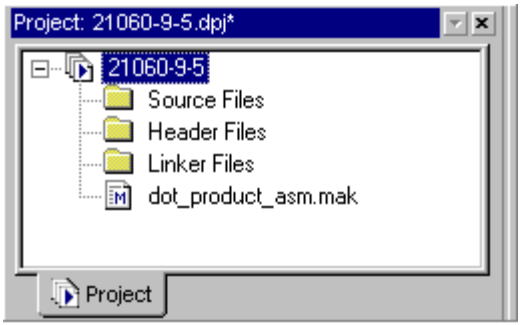

Figure 1-7. Makefile in Project Window

You can view a makefile's command line. To change the makefile's target, use the **Configuration** box, shown in Figure 1-8.

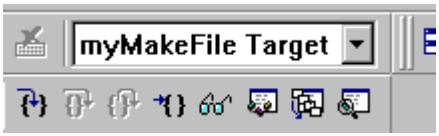

Figure 1-8. Makefile in Configuration Box

When you close a project, the Make commands and the target list associated with each makefile are serialized in the project (`.DPJ`) file.

Rules

You can enable only one makefile when building a project. If you enable more than one makefile, VisualDSP++ generates an error message. After you build your project with an external makefile, the executable file is not automatically loaded (even when this option is configured).

Output Window

Make command error messages and standard output appear in the **Output** window. Double-clicking on an error-message position opens the makefile in an editor window to the line of code causing the error.

Keywords in the makefile are syntax-colored.

Note: The error message format of `gmake` is parsed correctly when you double-click on an error message. If you use another make utility, the double-click feature does not function.

Example Makefile

An example of a makefile appears below.

```
# Generated by the VisualDSP++ IDDE
# Note:  Any changes made to this Makefile will be lost the next
# time the matching project file is loaded into the IDDE.  If you
# wish to preserve changes, rename this file and run it
# externally to the IDDE.
# The syntax of this Makefile is such that GNU Make v3.77 or
# higher is required.
# The current working directory should be the directory in which
# this Makefile resides.
# Supported targets:
#     Debug
#     Debug_clean
#     Release
#     Release_clean
# Define ADI_DSP if it is not already defined.  Define this
# variable if you wish to run this Makefile on a host other than
# the host that created it and VisualDSP++ may be installed in a
# different directory.
ifndef ADI_DSP
ADI_DSP=C:\Program Files\Analog Devices\VisualDSP
endif
# $VDSP is a gmake-friendly version of ADI_DIR
empty:=
```

```
space:= $(empty) $(empty)
VDSP_INTERMEDIATE=$(subst \,/,$(ADI_DSP))
VDSP=$(subst $(space),\$(space),$(VDSP_INTERMEDIATE))
# Define the command to use to delete files (which is different
# on Win95/98 and Windows NT/2000)
ifeq ($(OS),Windows_NT)
RM=cmd /C del /F /Q
else
RM=command /C del
endif
#
# Begin "Debug" configuration
#
ifeq ($(MAKECMDGOALS),Debug)
Debug : ./debug/mean.dxe
./debug/mean.doj :./mean.c ../../../include/stdio.h
$(VDSP)/cc21k -c .\Mean.c  -g  -proc ADSP-21062  -o
.\Debug\Mean.doj
./debug/benchmark.doj :./benchmark.asm
../../../include/asm_sprt.h ../../../include/def21060.h
$(VDSP)/easm21k.exe -proc ADSP-21062  -o .\Debug\benchmark.doj -g
.\benchmark.asm
./debug/mean.dxe :./debug/mean.doj ./debug/benchmark.doj
$(VDSP)/cc21k.exe .\Debug\Mean.doj .\Debug\benchmark.doj -proc
ADSP-21062 -L .\Debug -flags-link -od,.\Debug -o .\Debug\Mean.dxe
endif
ifeq ($(MAKECMDGOALS),Debug_clean)
Debug_clean:$(RM) ".\Debug\Mean.doj"
   $(RM) ".\Debug\benchmark.doj"
    $(RM) ".\Debug\Mean.dxe"
    $(RM) ".\Debug\*.ipa"
    $(RM) ".\Debug\*.opa"
    $(RM) ".\Debug\*.ti"

endif
# Begin "Release" configuration
#
ifeq ($(MAKECMDGOALS),Release)
Release : ./release/mean.dxe
./release/mean.doj :./mean.c
```

```
$(VDSP)/cc21k -c .\Mean.c  -O1  -proc ADSP-21062  -o
.\Release\Mean.doj
./release/benchmark.doj :./benchmark.asm
$(VDSP)/easm21k.exe -proc ADSP-21062  -o .\Release\benchmark.doj
.\benchmark.asm
./release/mean.dxe :./release/mean.doj ./release/benchmark.doj
$(VDSP)/cc21k.exe .\Release\Mean.doj .\Release\benchmark.doj
-proc ADSP-21062 -L .\Release -flags-link -od,.\Release -o
.\Release\Mean.dxe
endif
ifeq ($(MAKECMDGOALS),Release_clean)
Release_clean:
      $(RM) ".\Release\Mean.doj"
    $(RM) ".\Release\benchmark.doj"
    $(RM) ".\Release\Mean.dxe"
    $(RM) ".\Release\*.ipa"
    $(RM) ".\Release\*.opa"
    $(RM) ".\Release\*.ti"
endif
```

Project Configurations

By default, a project includes two configurations, Debug and Release, described in Table 1-6. In previous software releases, the term *configuration* was called "build type."

Table 1-6. Default Project Configurations

Configuration	Description
Debug	Builds a project that enables the use of VisualDSP++ debugging capabilities
Release	Builds a project with optimization enabled

Available configurations appear in the configuration box, which, by default, is located in the **Project** toolbar, as shown in Figure 1-9.

You cannot delete the Release or Debug configuration.

Figure 1-9. Configuration Box

Customized Project Configurations

You can add a configuration to your project. A customized project configuration can include various project options and build options to help you develop your project. Figure 1-10 shows a customized configuration (**Version2**) listed in the configuration box.

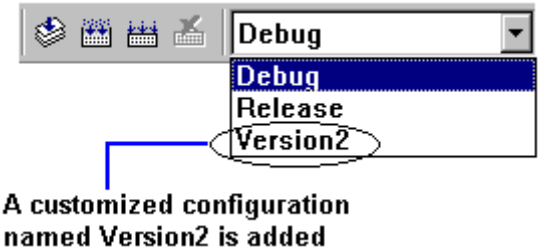

Figure 1-10. Selecting a Project Configuration

Project Build

The term *build* refers to the process of performing operations (such as preprocessing, assembling, and linking) on projects and files. During a build, VisualDSP++ processes project files that have been modified since the previous build as well as project files that include modified files.

A build differs from a *rebuild all*. When you execute the **Rebuild All** command, VisualDSP++ processes all the files in the project, regardless of whether they have been modified.

Building a project builds all outdated files in the project and enables you to make your program. An *outdated file* is a file that has been modified since the last time it was built or a file that includes (`#include`) a modified file. For example, if a C file that has not been modified includes a header file that has been modified, the C file is out of date.

VisualDSP++ uses *dependency* information to determine which files, if any, must be updated during a build.

Note the following:

- A file with an unrecognized file extension is ignored at build time.
- If an included header file is modified, VisualDSP++ builds the source files that include (`#include`) the header file, regardless of whether the source files have been modified since the previous build.
- File icons in the **Project** window indicate file status (such as excluded files or files with specific options that override project settings).

Build Options

You can specify options for the entire project and for individual files. Table 1-7 describes these build options.

Table 1-7. Build Options

Options	Description
Project-wide	Specify these options from a tabbed page (for example, **Compile** or **Link**) for each of the code development tools.
Individual-file	These settings override project-wide settings.
Custom-build step	For maximal flexibility, edit the command line(s) issued to build a particular file. For example, you might call a third-party utility.

File Building

Building a file compiles or assembles the file and locates and removes errors. You can build a single file or multiple files that you select.

The build process updates the selected source file's output (`.OBJ`) file and the output file's debug information. Building a single file is very fast. Large projects, however, may require hours to build.

If you change a common header file that requires a full build, you can build only the current file to ensure that your change fixes the error in the current file.

Batch Builds

Performing a batch build builds one or more build targets in the open project group. You must configure the batch build before you can build it.

A build target in a project group is formed by the combination of a project and a project configuration (such as a Release configuration). Refer to VisualDSP++ Help for details on configuring and running a batch build.

Pre-Build and Post-Build Options

Pre-build and post-build options are typically DOS commands that are executed before building a project and after a successful project build. These commands invoke external tools.

For example, you can use a post-build command to copy the final output file to another location on the hard drive or to invoke an application automatically.

Automatically copying files and cleaning up intermediate files after a successful build can be very useful.

Command Syntax

Place "`c:\windows\command.com /C`" at the beginning of each DOS command line. For example, to execute "`copy a.txt b.txt`", type:

```
c:\windows\command.com /C copy a.txt b.txt
```

The letter "`C`" after the slash character must be uppercase.

Project Dependencies

Dependency data determines which files must be updated during a build. The following are examples of dependency information.

```
..\.\.\include\cplus\cstddef
..\.\.\include\cplus\exception
..\.\.\include\cplus\new
..\.\.\include\cplus\xstddef
..\.\.\include\def21060.h
..\.\.\include\limits.h
..\.\.\include\cplus\stddef
..\.\.\include\stdio.h
..\.\.\include\string.h
..\.\.\include\VDK_Internals.h
..\.\.\include\VDK_Public.h
..\.\.\include\yvals.h
```

Project Window Rules

The **Project** window displays a project's files, as shown in Figure 1-11.

The following rules dictate how files and subfolders behave in the **Project** window's file tree.

- You can include any file in a project.
- Only one `.LDF` file is permitted.

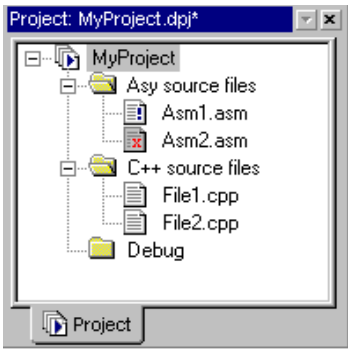

Figure 1-11. Example of Project Files

- You cannot add the same file into the same project more than once.
- Only one project (project node) is permitted.
- A file with an unrecognized file extension is ignored at build time.
- When you add a file to a project, the file is placed in the first folder configured with the same extension. If no such folders are present, an added file goes to the project level.

VisualDSP++ Help System

The VisualDSP++ Help system is designed to help you obtain information quickly. Use the Help system's table of contents, index, full-text search function, and extensive hyperlinks to jump to topics. Bookmark topics that you plan to revisit.

VisualDSP++ Help is comprised of multiple Help systems (`.CHM` files). Each file is identified with a book icon in the software installation's `Help` folder.

The majority of the Help system files are VisualDSP++ manuals and hardware documentation. These manuals are also available in PDF format (on the installation disk) for printing. Manuals are also available from Analog Devices as printed books.

Each window, toolbar button, and menu-bar command in VisualDSP++ is linked to a topic in Help. Other portions of the VisualDSP++ Help system provide procedures for configuring and using tools.

Some `.CHM` files support pop-up messages for dialog box controls (buttons, fields, and so on). These messages, which appear in little yellow boxes, comprise part of the context-sensitive Help in VisualDSP++.

For more information about the Help system, refer to "Online Help" on page A-51.

2 ENVIRONMENT

VisualDSP++ is an intuitive, easy-to-use user interface for programming Analog Devices processors. This chapter introduces the VisualDSP++ work environment, including the main window and debugging windows. Graphics are used to illustrate concepts and available window options.

Customize VisualDSP++ to meet your needs. Refer to VisualDSP++ online Help for "how to" information. This chapter is organized as follows.

- "Parts of the User Interface" on page 2-2

  This section introduces the application's main window (title bar, control menu, menu bar, toolbars, and status bar).

- "VisualDSP++ Windows" on page 2-14

  This section details the Project window, editor windows, and the Output window.

- "Debugging Windows" on page 2-50

  VisualDSP++ provides a multitude of debugging windows. This section focuses on each window, presenting its purpose, features, and benefits.

Parts of the User Interface

When you open VisualDSP++, the application's main window appears. Figure 2-1 shows an example of the VisualDSP++ main window.

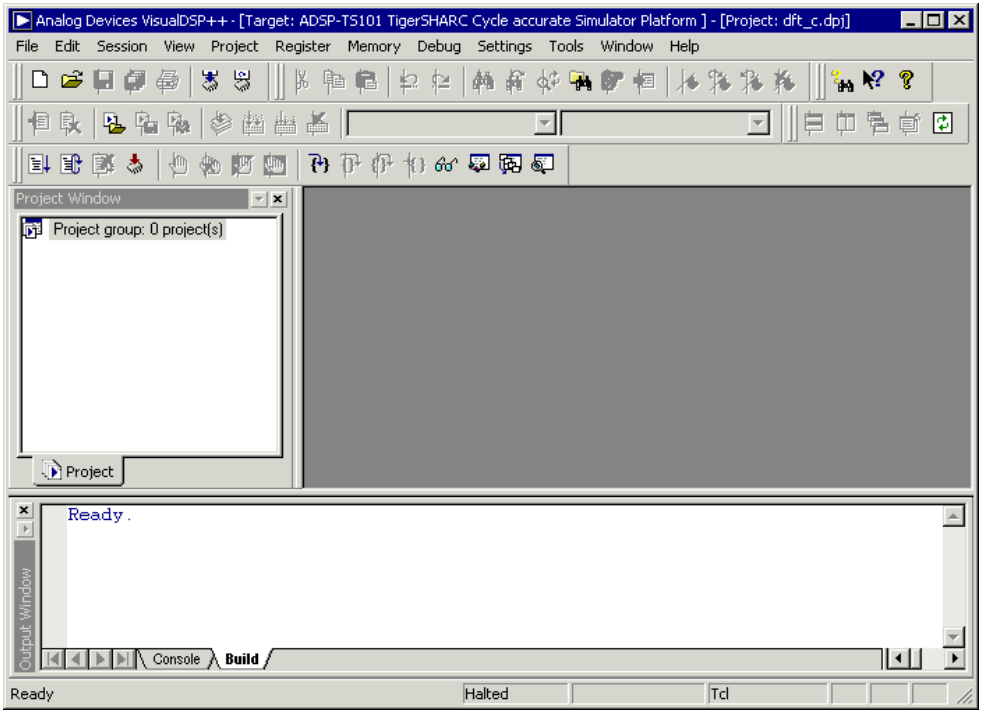

Figure 2-1. VisualDSP++ Main Window

This work area contains everything necessary to build, manage, and debug a project. You can set up preferences that specify the appearance of application objects (fonts, visibility, and so on). You can open project files by dragging and dropping them into the main window.

The VisualDSP++ main window includes these parts:

- Title bar and control menu
- Menu bar, toolbars, and status bar
- Project window
- Output window

VisualDSP++ also provides access to many debugging windows to facilitate project development. For more information, see “Debugging Windows” on page 2-50. You have to learn only one interface to debug all your processor applications.

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format, Debugging Information Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.

Title Bar

Figure 2-2 shows the different parts of the title bar.

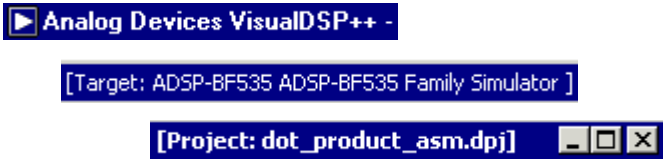

Figure 2-2. Title Bar (Split into Three Parts to Fit the Page)

The title bar includes these components:

- Control menu button
- Application name – Analog Devices VisualDSP++
- Name of the active target

- Project name
- File name (when an editor window is maximized in the main window)
- Standard Windows buttons

Clicking the control menu button opens the control menu, which contains commands for positioning, resizing, minimizing, maximizing, and closing the window. Double-clicking the control button closes VisualDSP++. The title bar right-click menu (Figure 2-3) and control menu (Figure 2-4) are identical.

Additional Information in Title Bars

A register window's title bar displays its numeric format (such as octal). An editor window's title bar displays the name of the source file.

Title Bar Right-Click Menus

A menu like the one in Figure 2-3 appears when you right-click within the VisualDSP++ title bar or within the title bar of a child (sub) window.

Figure 2-3. Right-Clicking in the VisualDSP++ Window's Title Bar

From the VisualDSP++ title bar's right-click menu, you can:

- Resize or move the application window
- Close VisualDSP++

Control Menu

Control menu (system menu) commands move, size, or close a window. Refer to

Figure 2-4. VisualDSP++ Control Menu

Program Icons

Click a program icon to open a control menu.

Program icon for the application and debugging windows

Program icon for editor windows

Placing the mouse pointer over a control menu command displays a brief description of the command appears in the status bar at the bottom of the application window.

Editor Windows

A floating editor window's control menu includes the **Next** command, which moves the focus to another window.

When an editor window floats in the main application window, its program icon resides at the left side of its title bar. When an editor window is maximized, the program icon resides at the left end of the menu bar. Editor windows are described on page 2-28.

Debugging Windows

Each debugging window has a control menu. You can open a debugging window's control menu only when the window is floating in the main window. For more information, see "Debugging Windows" on page 2-50.

Menu Bar

By default, the menu bar (Figure 2-5) appears directly below the application title bar. It displays menu headings, such as **File** and **Edit**.

Figure 2-5. VisualDSP++ Menu Bar

To display menu commands and submenus, click a menu heading. You can also run many menu bar commands by:

- Clicking toolbar buttons
- Typing keyboard shortcuts
- Right-clicking and choosing a command from a context menu

Toolbars and User Tools

A toolbar is a set of buttons. Run a command quickly by clicking a toolbar button.

Use toolbars to organize the tasks that you use most often. Position toolbars on the screen for fast access to the tools that you plan to use. VisualDSP++ includes standard (built-in) toolbars. You can create custom toolbars.

Built-In Toolbars

Table 2-1 shows the standard (default) toolbars.

Table 2-1. Built-In Toolbars

Name	Toolbar
File	
Edit	
Help	
Project	
Window	
Debug	
Multiprocessor	
User Tools	
Workspaces	

To obtain information about a tool, move the mouse pointer over the tool and press the **F1** key.

Toolbar Customization

By default, nine standard toolbars (Table 2-1) appear near the top of the application window, below the menu bar.

You can change the appearance of toolbars by:

- Moving, docking, or floating the toolbars
- Adding buttons to (or removing from) toolbars
- Displaying *cool look* buttons, *large buttons*, or both

You can also:

- Hide toolbars from view
- Add and delete custom-built toolbars

Toolbars: Docked vs. Floating

By default, toolbars are located under the application's menu bar, but you can move them to these locations:

- Over a docked window
- On the main window
- Anywhere on the desktop

A toolbar attached to a window is called a *docked* toolbar. You can tell when a toolbar is going to dock by the size and shape of its moving outline as you drag it. Its outline becomes slightly smaller than its floating outline. To prevent a toolbar from docking, press and hold the **Ctrl** key while dragging the toolbar to a new location.

A toolbar can be detached from a window and moved to another location anywhere on the desktop. A *floating* toolbar is a stand-alone window, as it is not docked. A docked toolbar does not show its name; however, a floating toolbar displays its title.

Figure 2-6 shows a floating Help toolbar.

Figure 2-6. Floating Help Toolbar

Toolbar Button Appearance

You can specify the appearance of the toolbar buttons. Two options, *cool look* and *large buttons*, provide different button appearances.

The *cool look* option includes a pair of vertical bars on the toolbar's left side, but removes the square box from each button. Vertical bars visually separate toolbar buttons into groups (toolbars).

The *large buttons* option increases the size of each button.

Table 2-2 shows how small and large buttons appear with the *cool look* option turned off (disabled) and on (enabled).

Table 2-2. Toolbars in Different Viewing Options

Option Settings	Docked	Floating
Cool look – Off Large buttons - Off		
Cool look – On Large buttons – Off		
Cool look – Off Large buttons – On		
Cool look – On Large buttons – On		

Toolbar Shape

You can change the shape of a floating toolbar. Table 2-3 shows two toolbar shapes.

Table 2-3. Toolbars in Two Orientations

Horizontal	Vertical

Depending on the number of tools in the toolbar, you can create other length and width arrangements.

Toolbar Rules

When working with toolbars, be aware of these rules:

- You can customize a built-in toolbar (for example, by removing a button from the **File** toolbar), but you cannot delete a built-in toolbar. You can reset the buttons in a built-in toolbar to their original default settings.
- You can change the name of a user-defined toolbar, but not the name of a built-in toolbar. For example, the **File** toolbar cannot be changed to a different name.

Refer to VisualDSP++ online Help for details.

User Tools

Save time running commands by configuring user tools. A maximum of ten user tools may be configured.

A user tool runs a command, which can:

- Contain parameters to launch an application
- Be a script command

Access configured user tools from the **Tools** menu or from the **User Tools** toolbar, as shown in Figure 2-7.

Figure 2-7. Default User Tools

When a user tool is configured, its menu name (label) appears in the **Tools** menu. The label also appears when you move the mouse pointer over a user tool button.

Status Bar

The status bar, which is located at the bottom of the main application window, provides various informational messages. Figure 2-8 shows examples of information displayed on the status bar.

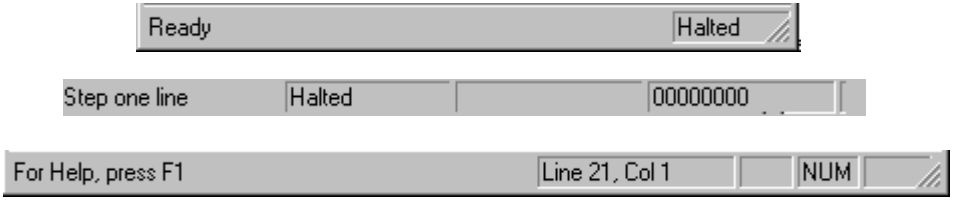

Figure 2-8. The Status Bar's Appearance Depends on Context

The type of information that appears in the status bar depends on your context (what you are doing).

- Moving the mouse pointer over a toolbar button or a menu bar command displays a brief description of the button or command.
- Halting program operation with a **Halt** command displays the address where the program halted.
- When using script commands, the status bar provides information, such as when the menu item has focus.

While editing a file, the right side of the status bar displays editor window information, as described in Table 2-4.

Table 2-4. Status Bar Information While Editing

Item	Indicates
Line ###	Cursor current line number
Col ###	Cursor current column number
CAP	The keyboard's **Caps Lock** key is latched down
NUM	The keyboard's **Num Lock** key is latched down
SCRL	The keyboard's **Scroll Lock** key is latched down

VisualDSP++ Windows

From the application's main window, you can open the **Project** window, editor windows, the **Output** window, and various debugging windows.

Project Window

To open a **Project** window, choose **View** and **Project Window**. Figure 2-9 shows a **Project** window with VDK enabled.

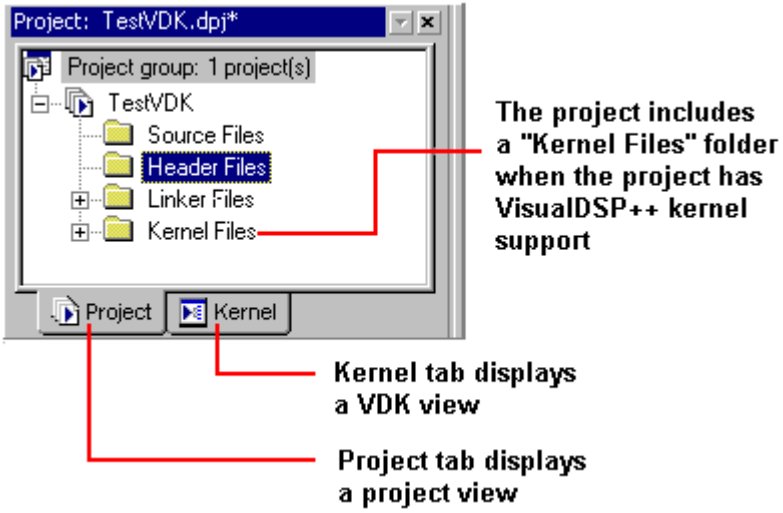

Figure 2-9. Example Project Window With Kernel Tab

An additional folder, titled **Generated Files**, may be present when a project includes startup files.

The **Project** window can include two sub-tabs:

- The **Project** tab

, which is always available, displays a hierarchal representation of a debug session's projects, folders, files, and dependencies.
- The **Kernel** tab

appears when VDK is enabled for a project. It displays VDK-related information.

Project View

The **Project** view displays a project group, which may contain any number of projects. Only one project, however, is active at a time. Nodes are arranged in a hierarchy similar to the file structure in Windows Explorer.

Figure 2-10 shows an example of information that displays in **Project** view.

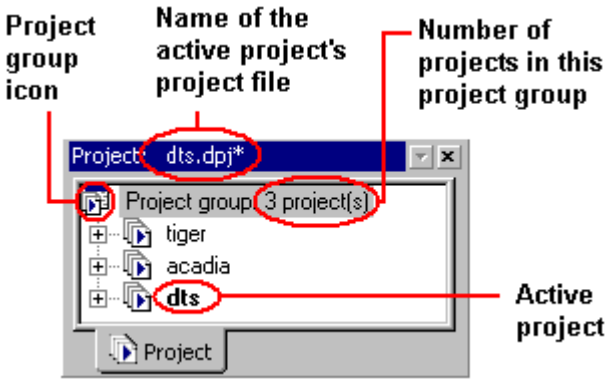

Figure 2-10. Project View

Project Dependencies

A project may depend on other projects. The icon indicates dependency and identifies the dependency. Building a project also builds the sub-projects on which your project depends.

Figure 2-11 shows how project dependencies appear in the **Project** view.

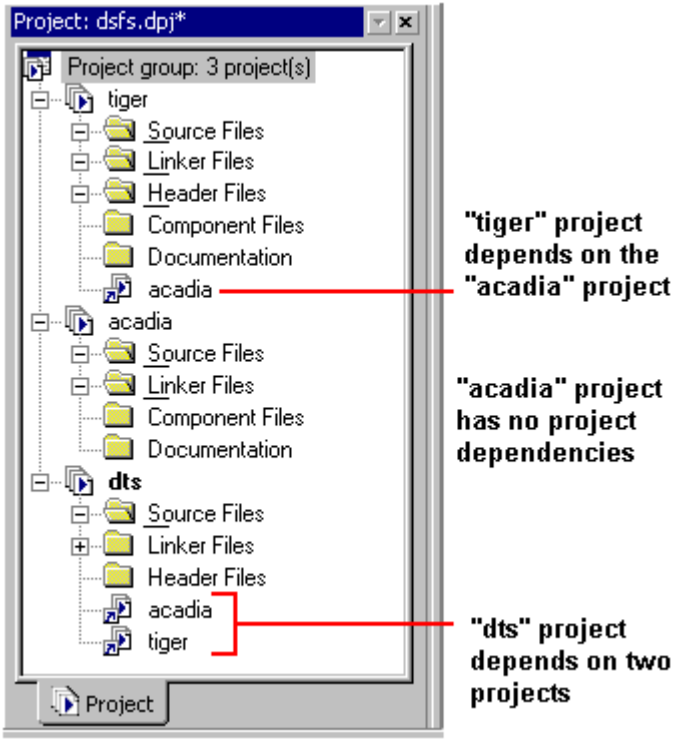

Figure 2-11. Projects Dependencies Indicated in the Project View

Project Nodes

The **Project** window comprises the types of nodes described in Table 2-5.

Table 2-5. Types of Nodes in the Project Window

Node	Icon	Description
Project group		Only one project group permitted in a debug session
Project		Multiple projects permitted, but only one is active (indicated with bold typeface)
Folder		Closed folder
		Opened folder revealing its contents
File		File that uses project settings
		File whose options differ from the project options
		File excluded from the current configuration
		Enabled (active) makefile
Project dependency		Project on which another project depends

Project Page Right-Click Menus

Right-click menus (also called context menus or pop-up menus) operate on **Project** window objects (the project group, projects, folders, and files). These menus provide fast access to many menu bar and toolbar commands. The commands available in a right-click menu depend on context (the selected object).

Project window right-click menus offer these standard commands:

- **Allow Docking** (dock the **Project** window to the frame)
- **Hide** (remove the **Project** window from view)
- **Float in Main Window**

Project Group Icon Right-Click Menu

The project group icon ( ) right-click menu (Figure 2-12) provides a project group context from which to:

- Create a new project
- Open a project and add it to the project group
- View the project group's properties

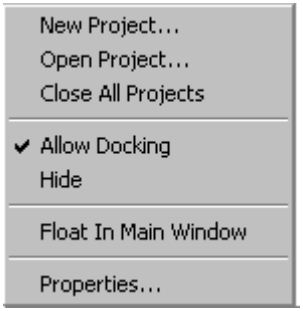

Figure 2-12. Project Group Icon's Right-Click Menu

Project Icon Right-Click Menu

The project icon ( ) right-click menu (Figure 2-13) provides a project context from which to:

- Build the project
- Clean (delete intermediate and target files)
- Specify the active project
- Add folders and files
- View and specify project options
- View project properties

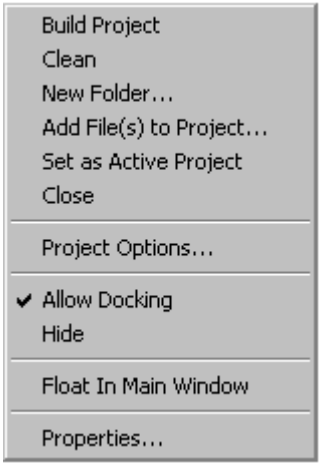

Figure 2-13. Project Icon's Right-Click Menu

Folder Icon Right-Click Menu

The selected folder icon (or) right-click menu (Figure 2-14) provides a "container" context from which to perform these "local" operations:

- Add or delete a folder
- Add files to the folder
- View folder properties

Figure 2-14. Folder Icon Right-Click Menu

File Icon Right-Click Menu

The selected file icon (or or) right-click menu (Figure 2-15 on page 2-21) provides a file context from which to:

- Open the selected file for editing
- Build the file
- Remove the file from a project
- Specify options for the file
- View the file's properties

Figure 2-15. File Icon Right-Click Menu

File icon commands apply to the selected file in the **Project** window, not to a source file in an editor window.

Project Folders

Project window folders (and) organize files within a project. You can specify properties for folders.

Folders can be nested to any depth. Folders carry no attributes to the build process, as they do not reflect the file system. Folders do not appear in directory listings, as in Windows Explorer.

When you add files to the project tree with automatic file placement, each file is placed in the first folder configured with the same file extension. After automatic placement, you can manually move a file anywhere.

To move a file out of one folder and into another folder, select the file and drag it onto the other folder.

Project Files

In the **Project** window, files are represented by the icons in Table 2-6.

Table 2-6. File Icons in the Project Window

Icon	Description
	Files that use project options
	Files that use options that differ from project options
	Files excluded from the current configuration
	Enabled (active) makefile

The files appear in an expandable and collapsible node tree.

Source files are the C/C++ language or assembly language files in your project. Source files provide the project with code and data. You can add, delete, and modify source files.

Each project must include one `.LDF` file, which contains command input for the linker. If an `.LDF` file is not included in the project, the project is built with a default `.LDF` file.

A project can also include data files and header files.

Project Window Icons for Source Code Control (SCC)

Icons in the **Project** window indicate source code control (SCC) status. An icon with a green check mark (✔) indicate that the file is under SCC and is checked in. An icon with a red check mark (✔) indicate that the file is under SCC and is checked out. Files that are not connected to a controlled copy under SCC do not display a check mark.

Table 2-7 shows examples of file icons used to indicate SCC status.

Table 2-7. SCC Status Icons

Icon	Description
	File is under SCC and is checked in
	File is under SCC and is checked out
	Project file is checked out
	File includes a file-specific build command and is checked out
	Makefile is checked out
	File is excluded from the build and is checked out

File Associations

VisualDSP++ associates the file extensions in Table 2-8 as the input to particular code development tools.

Table 2-8. File Associations

Tool	File Extensions
Compiler	.C, .CPP, and .CXX
Assembler compiler:file associations for tools	.ASM, .S, and .DSP
Linker	.LDF, .DLB, and .DOJ

VisualDSP++ is case insensitive to file extensions.

Automatic File Placement

Automatic file placement is a feature that enables you to drag and drop files into designated folders on the **Project** page in the **Project** window. This saves time when you add files to a project.

File placement rules and the folder properties that you specify determine where files are placed. By default, project folders are associated with the file extensions listed in Table 2-9.

Table 2-9. Default Files Associations in Project Folders

Folder	Default Associations
Source Files	.C, .CPP, .CXX, .ASM, .DSP, .S
Header Files	.H, .HPP, .HXX
Linker Files	.LDF, .DLB, .DOJ
Kernel Files	.VDK

File Placement Rules

The following rules dictate file placement when you add files to a project.

- Dragging and dropping files

  When you drag and drop a file onto the **Project** page, the file is added to the first folder associated with the file's extension. The **Project** page accepts dragged files only when a project is opened.

- Using menu commands to add files

  Files are added to the folders that you select on the **Project** page. If you add a file to a project that has no folders, the file is added at the project level (root level).

  If you select the project node or a file node, the file is added to the first folder associated with the file's extension.

Example

You create a folder labeled "C Source Files" and specify it with `.C`, `.CPP`, and `.CXX` file extensions. You create a second folder labeled "`Asm Files`" and associate it with `.ASM` files.

If you drag three files (`file1.cpp`, `file1.asm`, and `file2.c`) into the **Project** window, `file1.cpp` and `file2.c` go into the `C Source Files` folder, and `file1.asm` goes into the `Asm Files` folder.

After automatic file placement, you can manually move a file anywhere by selecting and dragging the file.

Kernel Page

The **Kernel** tab of the **Project** window is available only for VDK-enabled projects.

From the **Kernel** page, you can add, modify, and delete kernel elements such as thread types, priorities, semaphore, and events. VisualDSP++ automatically updates `vdk_config.cpp` and `vdk_config.h` to reflect the changes made on the **Kernel** page.

The example in Figure 2-16 shows an expanded view of the elements on the **Kernel** page for a VDK-enabled project.

For information about VDK, refer to the *VisualDSP++ 4.0 Kernel (VDK) User's Guide*.

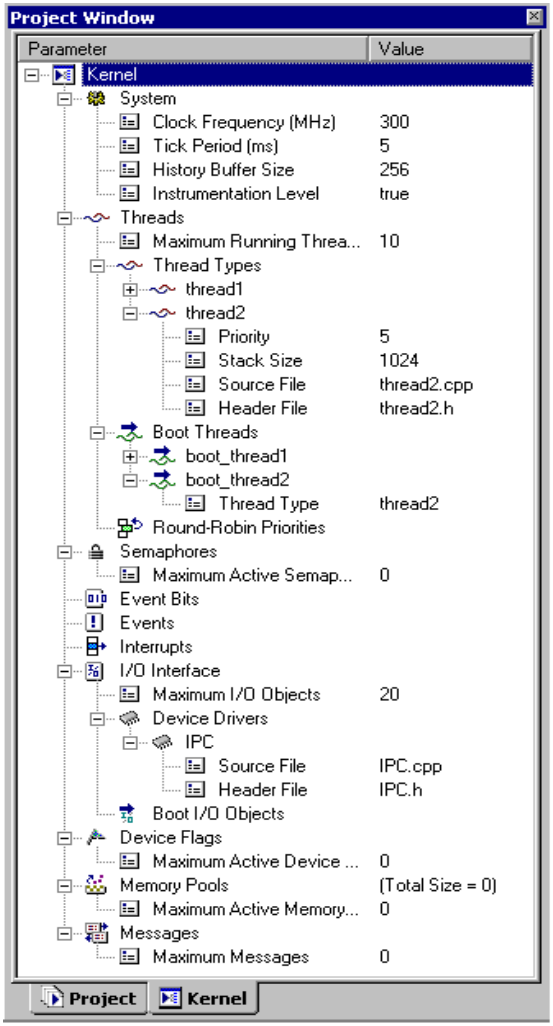

Figure 2-16. Expanded View of Elements on the Kernel Page

Editor Windows

Use editor windows to view and edit files. Open as many editor windows as you like from the **Project** window by double-clicking on a file or by choosing **Open File** from a file's right-click menu.

Figure 2-17 shows items that can be customized in editor windows.

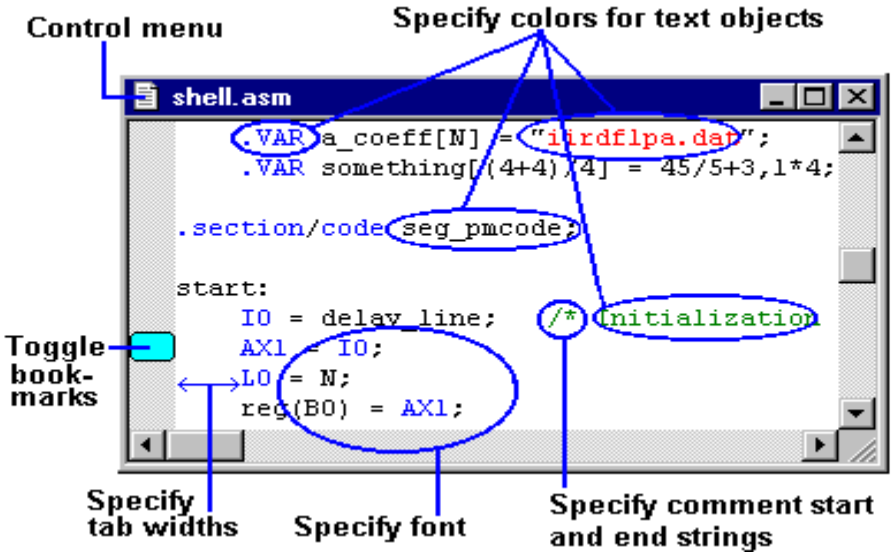

Figure 2-17. Customizable Items in Editor Windows

Editor windows support:

- User-defined color-coded comments, strings, keywords, and tab settings (syntax coloring)
- Two viewing modes: source mode and mixed mode
- Printing, print preview, and user-defined headers and footers
- Bookmarking

- Finding/replacing with wrap-around search and regular expression matching
- Going to a line number (and the display of line numbers)
- Jumping to the next or previous syntax error
- Copying, cutting, pasting, undoing, and redoing functions. Each open file has a deep stack (500+ items).
- Tabs for fast switching between source files
- Location of matching brace characters and auto-positioning of brace characters (to line up with the preceding opening brace)
- Opening header files by right-clicking on `#include` statements
- Dragging-and-dropping highlighted sections of text (usually a valid source statement) to an open Expressions window. When dropped, the text is automatically added to the window and is evaluated.
- Running scripts

Many of these features are described next. Refer to VisualDSP++ online Help for how-to information.

Editor Window Symbols

The gutter (left margin) in an editor window displays icons to indicate breakpoints, bookmarks, and the current position of the program counter (PC). Table 2-10 describes these icons.

Table 2-10. Editor Window Symbols

Symbol	Indicates
	Current source line to be executed (PC location)
	Enabled breakpoint
	Disabled breakpoint
	Bookmark

Bookmarks

Bookmarks are pointers in editor windows. Place a bookmark in a location to return to it quickly at a later time.

Syntax Coloring

Specifying colors can help you locate information in the types of files listed in Table 2-11.

Table 2-11. File Types That Support Syntax Coloring

File Type	File Extension
Assembly	.ASM
C	.C
Linker Description Files	.LDF
C++	.CPP
Header	.H
Script	Various extensions, such as .JS and .VBS

Viewing Modes: Source Mode vs. Mixed Mode

Specify an editor window to display in source mode or mixed mode.

Source Mode

Source mode, as shown in Figure 2-18, displays C code only.

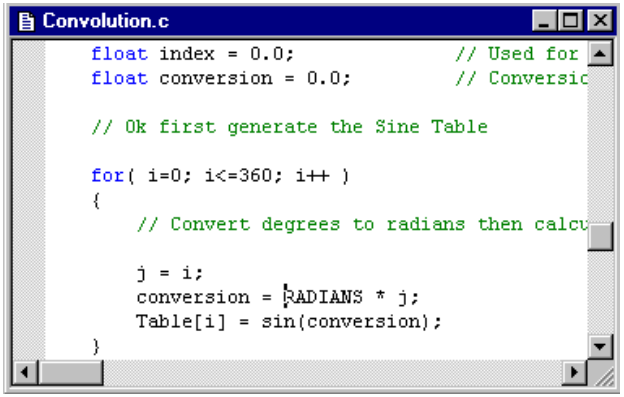

Figure 2-18. Example: Editor Window in Source Mode

Mixed Mode

Mixed mode displays assembled code immediately after the corresponding C code. The assembly code takes a specified color.

Note:

- To be viewable in mixed mode, the source file must be compiled with debugging information enabled.
- The display of pipeline symbols can be enabled or disabled in mixed mode.

Figure 2-19 shows an example of the mixed mode format.

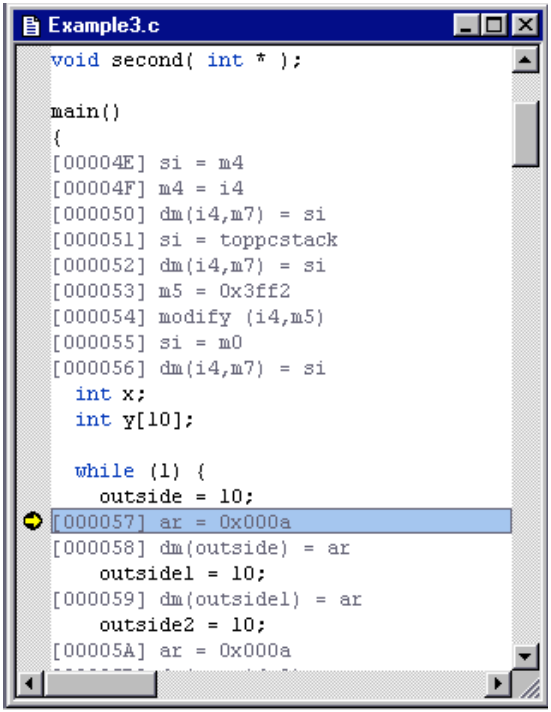

Figure 2-19. Example: Editor Window in Mixed Mode

Editor Tab Mode

Editor Tab mode provides an alternative, tab-based user interface for managing multiple source files in editor windows. When this mode is enabled via the **View** menu, a tab for each open source file appears at the bottom of the editor window. Clicking on tabs switches between files.

Figure 2-20 shows an editor window with the **Editor Tab** option enabled.

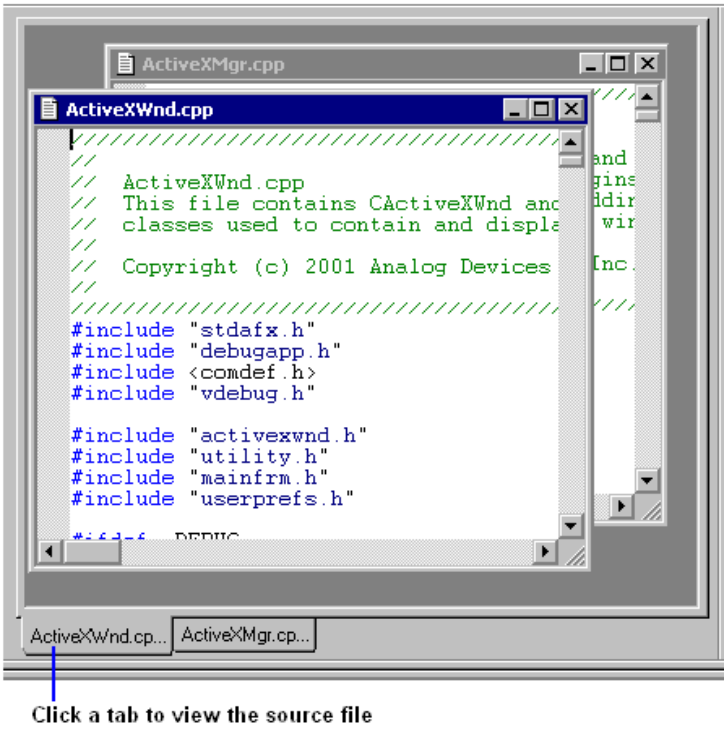

Figure 2-20. Switching Between Editor Windows Using Editor Tab Mode

Context-Sensitive Expression Evaluation

When a .DXE program has been loaded for debugging, you can evaluate expressions in an editor window.

As you move the mouse pointer over a variable, with the pointer still on top of the variable, VisualDSP++ evaluates the variable. If the variable is in scope, the value appears in a tool tip window.

Viewing an Expression

Expressions can be viewed in different ways. When the editor window is in mixed mode, view an expression by moving the pointer over a register in an assembly instruction. The register contents are displayed in a tool tip.

Highlighting an Expression

Highlight an expression in the editor window and then move the pointer on top of the highlighted expression to display its value in a tool tip.

Right-Click Menu

The editor window's right-click menu provides the commands shown in Figure 2-21.

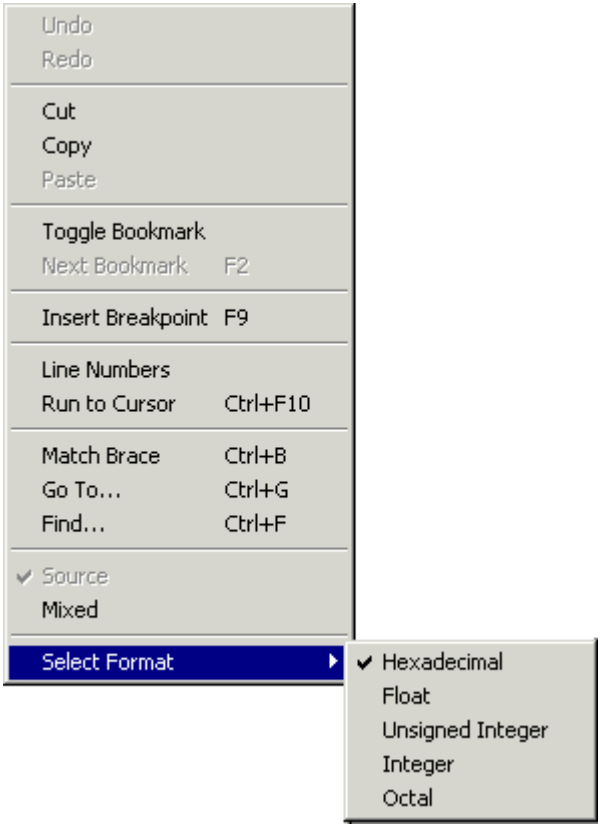

Figure 2-21. Example: Editor Window Right-Click Menu

The available formats under **Select Format** depend on the target processor. An additional command, **Source Script**, is available when you are editing a script.

Output Window

The **Output** window:

- Displays standard I/O text messages such as file load status and error messages
- Displays build status information for the current project build
- Provides access to errors in source files
- Acts as a scripting interface

The example **Output** window in Figure 2-22 shows build status information. Display the **Output** window by choosing **View** and **Output Window**.

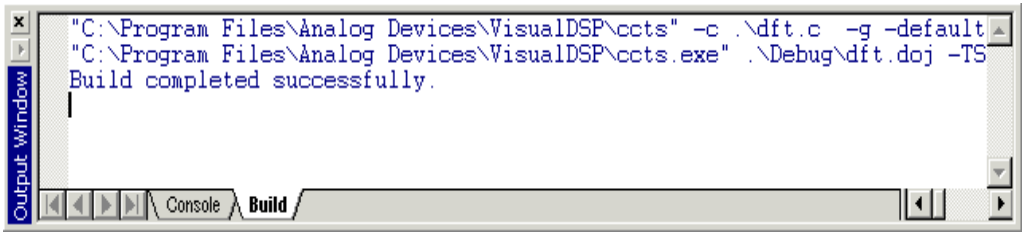

Figure 2-22. Viewing Build Status Information in the Output Window

Viewing Error Message Details

Each tool error message has associated explanatory text. You can view the information in the Help window.

To view an error message's details, from the **Output** window's **Build** page, use the PC's mouse to select the six-character error identifier (for example, cc0276). Selection is sometimes easier from right to left.

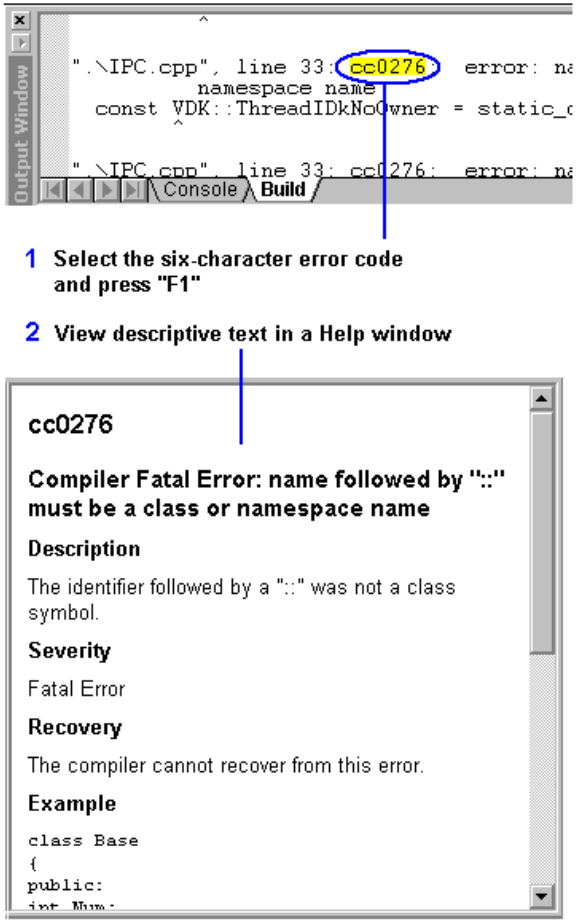

Figure 2-23. Example: Viewing Details of an Error Message in Help

Output Window Tabs

The **Output** window's two tabs, **Console** and **Build**, provide different information and capabilities.

Build Page

The **Build** page (Figure 2-24) displays error messages generated during a build.

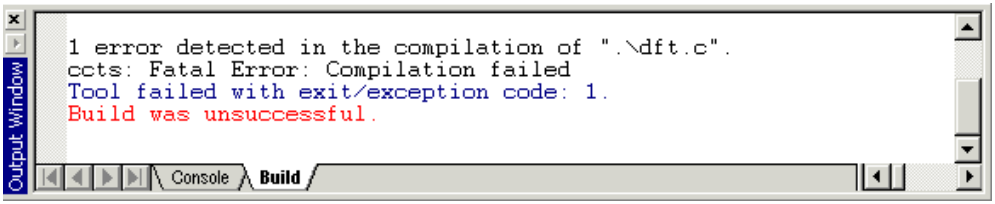

Figure 2-24. Example: Error Messages in the Output Window

Choosing **Next Error** or **Prev Error** from the **Edit** menu allows you to scroll through error messages.

Double-clicking on an error message displays the offending code in an editor window. Refer to "Viewing Error Message Details".

By default, VisualDSP++ output is blue and tool output is black. You can modify these colors via the **Preferences** dialog box.

Console Page

The **Output** window's **Console** page (Figure 2-26 on page 2-46) allows you to:

- View VisualDSP++ or target status error messages
- View STDIO output from C/C++ programs
- View I/O (streams) messages

- Scroll through previous commands by pressing the keyboard's up arrow (↑) and down arrow (↓) keys
- Perform multiline selection, copy, paste, and clear
- Issue script commands and view script command output
- Auto-complete script commands
- Execute a previously issued script command by double-clicking on the command
- Enter multiline script commands by adding a backslash character (\) to the end of a statement
- Use bookmarks
- Toggle a bookmark by pressing **Ctrl+F2**
- Move to the next bookmark by pressing the keyboard's **F2** key

All text that displays on the **Console** page is written to the VisualDSP++ log file.

Output Window Error Messages

The code development tools that perform batch processing can produce error and warning messages when returning a result. These informational messages appear on the **Build** page in the **Output** window.

Every error is identified with a unique six-character code, such as `pp0019`, that is consistent between versions of VisualDSP++. Error descriptions include an explanation of the condition that caused the error and a suggested remedy to fix the problem. Where applicable, error messages include the source file's name and the line number of the offending code.

Error Message Severity Hierarchy

Each error message has one or more severity levels.

Table 2-12. Error Message Severity Levels

Severity Level	Description
Fatal error	Identifies errors so severe that further processing of the input is suspended. Fatal errors are sometimes called catastrophic errors.
Error	Identifies problems that cause the tool to report a failure. An error might allow further processing of the input to permit the reporting of additional problems to be reported.
Warning	Identifies situations that do not prevent the tool from processing the input, but may indicate potential problems
Remark	Provides information of possible interest

You can change the severity level of an error marked “discretionary”. The severity level of an error marked “non-discretionary” cannot be changed.

Syntax of Help for Error Messages

In VisualDSP++ online Help, each error message can include several parts. The available information depends on the tool and the message.

To view all the details, you must view the error message in the Help system window. If you run a tool from a command-line interface (such as a **Command Prompt** window or **MS-DOS Prompt** window), the error message shows only the ID code, error text, and error location.

Table 2-13 describes the syntax for error message help.

Table 2-13. Syntax for Error Message Help

Part	Description
Identification Code	Six-character code, unique to the error. The first two characters identify the tool: • ar (archiver) • cc (compiler) • ea (assembler) • el (expert linker) • id (IDDE) • li (linker) • pp (preprocessor) • si (simulator)
Error Text	Text that appears after the identification code in the **Output** window
Description	Detailed description of the error
Severity	The degree of hardship imposed by the error. Some messages can take more than one severity level. You can change the severity level of an error marked “discretionary.” You cannot change the severity level of errors marked “non-discretionary.”
Recovery	Extra information, provided only if applicable
Example	Example code
How to Fix	The remedy for correcting the error
Related Information	Link(s) to more information

Promoting, Demoting, and Suppressing Error Messages

You can change the severity level of an error marked “discretionary.” Refer to the tools documentation for command-line switches that override error message severity. The **Project Options** dialog box provides options you can use to override severity.

A discretionary message can be promoted, demoted, or suppressed. For example, you might promote a remark or warning to an error, or you might decide to demote an error to a warning or remark.

Say, for example, that a condition in the input is crashing the tool. You could restrict the severity level of the problem to report an error (instead of a fatal error).

Another way to suppress the reporting of an individual error message is to use pragmas in the input source via the tool’s command line. For more information about pragmas, refer to your processor’s VisualDSP++ 4.0 C/C++ Compiler and Library manual.

The following examples demonstrate how to promote, demote, and suppress messages. The source file, `test.c`, is being compiled.

```
$#include <stdio.h>
int foo(void)
{
printf("In foo\n"); // doesn't return a value

int main(void)
{
int x; // no initial value
printf("x = %d\n", x);
return foo();
}
```

- **Example 1: Compiling from the Command Line (Interface)**

  Compiling `test.c` yields two warning messages (cc0117 and cc0549):

```
$ cc21k -c test.c
"test.c", line 5: cc0117: {D} warning: non-void function
"foo" should return a value
}
^
"test.c", line 10: cc0549: {D} warning: variable "x" is
used before its value is set
printf("x = %d\n", x);
^
build completed successfully
```

  Notice that the compiler appended the letter "`D`" to each warning message, indicating that the message is discretionary.

- **Example 2: Promoting Warnings to Errors**

  For example, typing `$ cc21k -c test.c -Werror 549` in a command window promotes one of the two warnings (cc0549) to an error.

```
$ cc21k -c test.c -Werror 549
"test.c", line 5: error cc0117: {D} warning:  non-void
function "foo" should return a value
}
^
"test.c", line 10: cc0549: {D} error: variable "x" is
used before its value is set
printf("x = %d\n", x);
^
1 error detected in the compilation of “test.c”.

cc21k: Fatal Error: Compilation failed
```

- **Example 3: Demoting Messages to Remarks**

  You can demote messages to remarks. By default, the compiler does not display anything less significant than a warning.

  The `-Wremarks` flag in the following command outputs the two warnings plus additional remarks.

  ```
  $ cc21k -c test.c -Wremarks
  ```

  The `-Wremark 549,117` flag in the following command demotes two specific messages to remarks. The command produces no output because all the messages are changed to remarks, which are not displayed.

  ```
  $ cc21k -c test.c -Wremark 549,117
  ```

  The following command changes the two warnings to remarks and then displays all seven remarks.

  ```
  $ cc21k -c test.c -Wremark 549,117 -Wremarks
  ```

- **Example 4: Suppressing Messages**

  The following command suppresses two specific warning messages. The command outputs five remarks, but the two warnings do not display even though the `-Wremarks` flag requests all the remarks.

  ```
  $ cc21k -c test.c -Wsuppress 549,117 -Wremarks
  ```

Suppressing Compiler Warnings and Remarks

You can suppress compiler warnings as well as compiler remarks.

You cannot suppress compiler warnings without also suppressing remarks.

From the **Compile** page (**Warning** subpage) of the **Project Options** dialog box, you can specify warning options.

Log File

The VisualDSP++ log file contains all the status and error messages that have been written to the **Output** window's **Console** page.

Figure 2-25 shows a sample log file.

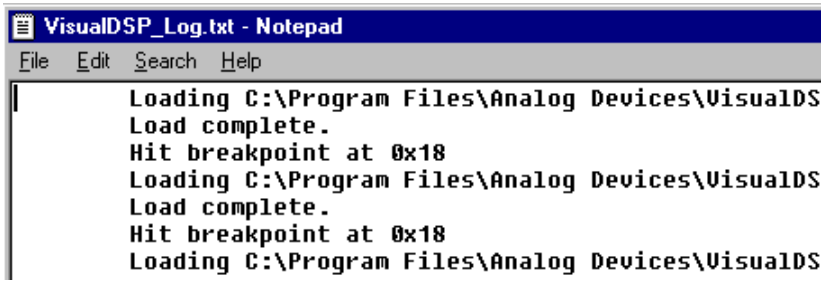

Figure 2-25. Portion of a Sample Log File

Messages are saved to the log file, `VisualDSP_Log.txt`, which, by default, is located in the `<install_path>\Data` directory.

All sessions append to the log file. Occasionally, open the file and delete parts of it (or all of it) to conserve disk space.

Output Window Customization

You can specify preferences that:

- Configure **Output** window fonts and colors
- Enable command auto-completion
- Display file names only while building (hide complete command lines)

By default, the **Output** window resides at the bottom of the main application window. You can resize or move the **Output** window to a different portion of the screen by dragging it to the selected location. You can dock, hide, or float the window.

The **Output** window's **Console** page interacts with script engines. All script input and output is sent to the **Console** page, as shown in Figure 2-26.

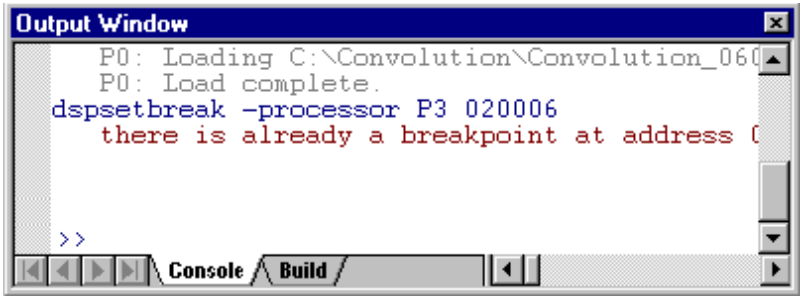

Figure 2-26. Messages in the Output Window's Console Page

These messages are saved to the log file, `VisualDSP_Log.txt`, which is located in the installation's `Data` directory.

Right-Click Menu

The **Output** window's right-click menu is shown in Figure 2-27.

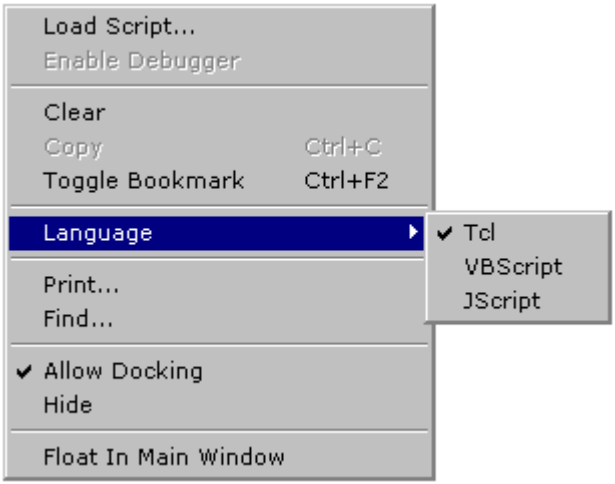

Figure 2-27. Output Window's Right-Click Menu

This menu enables you to:

- Load a script or enable the debugger
- Clear the text in the window or copy selected text
- Toggle bookmarks
- Select a scripting language
- Print or find text in the window
- Dock, hide, or float the window. (To display the hidden window, choose **Output Window** from the **View** menu.)

Script Command Output

Scripts provide a powerful means of developing full-blown test applications of processor systems. VisualDSP++ includes a language-independent scripting host that uses the Microsoft ActiveX® script host framework. This scripting host permits the use of multiple scripting languages that conform to the Microsoft ActiveX script engine.

The main benefit of calling scripts in these languages is that they have support for COM scripting, which allows access to the VisualDSP++ Automation API. VisualDSP++ supports the following Microsoft ActiveX script engines (languages):

- Visual Basic® (Scripting Edition)
- JScript®

The Tool Command Language (Tcl) interpreter included with VisualDSP++ is not a Microsoft ActiveX script engine. VisualDSP++ permits the use of other script engines (languages) that are not supported by Analog Devices technical support.

Script output is logged to `VisualDSP_Log.txt` for viewing and analysis. By default, this file is located in the installation's `Data` directory.

In the **Output** window's **Console** view, you can:

- Issue script commands and view script command output

  For more information about issuing script commands, refer to "Extensive Scripting" on page A-39.

- Enable the Microsoft Script Debugger

  Right-click in the **Output** window and choose **Enable Debugger**. The debugger steps through code, sets breakpoints, and so on. Once enabled, the debugger stops on the first error encountered in the script.

Although most script engines (languages) support this option, some may not. Consult the script engine's documentation for further details on whether it supports the debugging interfaces within the Microsoft ActiveX script engine framework.

- Specify the scripting language

  Right-click in the **Console** view and select a language from the list of scripting languages installed on your machine.

  The name of the current scripting language appears in the status bar at the bottom of the VisualDSP++ main window, as shown in Figure 2-28.

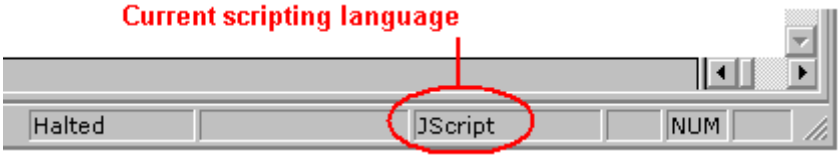

Figure 2-28. Scripting Language Displayed in Status Bar

- Load a script

  You can load a script by selecting **Load Script** from the **File** menu, from the **Console** view's right-click menu or the editor window's right-click menu. The script loads and runs until it finishes running or until you halt the script by choosing **Halt Script** from the **Debug** menu.

The **Console** view supports script command auto-completion, which you can enable on the **General** page of the **Preferences** dialog box, accessible via the **Settings** menu.

The VisualDSP++ installation directory includes example scripts in the "`Scripting Examples`" folder located under the processor family name (for example, `21k`) and the `Examples` folder.

Debugging Windows

VisualDSP++ provides debugging windows to display program operation and results. Table 2-14 describes these windows.

Table 2-14. Debugging Windows

Window	Provides
Output (on page 2-36)	A **Console** page that displays standard I/O text messages such as file load status, and error messages and streams, and a **Build** page that displays build messages. You can interactively enter script commands and view script output.
Editor (on page 2-28)	Syntax coloring, context-sensitive expression evaluation, and status icons that indicate breakpoints, bookmarks, and the current PC position
Disassembly (on page 2-52)	Code in disassembled format. This window provides fill and dump capability.
Expressions (on page 2-57)	The means to enter an expression and see its value as you step through program execution
Trace (on page 2-60)	A history of processor activity during program execution, including buffer depth (instruction lines), cycle count, and instructions executed such as memory fetches, program memory writes, and data/memory transfers (SHARC processors only)
Locals (on page 2-62)	All local variables within a function. Use this window with **Step** or **Halt** commands to display variables as you move through your program.
Linear Profiling Results (on page 2-63)	(Simulation only) Samples of the target's PC register taken at every instruction cycle, which provides an accurate picture of where instructions were executed. Linear profiling is much slower than statistical profiling.
Statistical Profiling Results (on page 2-63)	(JTAG emulation only) Random samples of the target processor's program counter (PC) and a graphical display of the resulting samples, showing where the application spends time
Call Stack (on page 2-72)	A means of moving the call stack back to the previous debug context
Register (on page 2-86)	Current values of registers. You can change register contents and change the number format.

Table 2-14. Debugging Windows (Cont'd)

Window	Provides
Custom Registers (on page 2-89)	Current values of registers. Select the registers that you want to monitor.
Memory (on page 2-72)	A view of processor memory. Similar number format and edit features as register windows, plus fill and dump capability.
BTC Memory (on page 2-78)	A view of background telemetry channel contents in real time. The window displays the contents of the address that you want to see. (SHARC and Blackfin emulator sessions only)
Memory Map (on page 2-85)	The memory map of the selected processor
Plot (on page 2-115)	A graphical display of values from memory addresses. The window supports linear and FFT (real and complex) visualization modes and allows you to export an image to a file, the clipboard, or to a printer.
Multiprocessor (on page 2-90)	Current status of each processor in a multiprocessor system (emulator sessions only). This window allows you to define and manage groups of processors for synchronous multiprocessor commands.
Pipeline Viewer (on page 2-94)	Display of instructions in the pipeline and event details (TigerSHARC and Blackfin processors only)
Cache Viewer (on page 2-99)	Analysis of an application's use of cache, which is helpful in optimizing application performance
VDK State History (on page 2-111)	(VDK-enabled projects only) History buffer of threads and events
Target Load (on page 2-114)	(VDK-enabled projects only) Percent of time the target spent in the idle thread
VDK Status (on page 2-109)	(VDK-enabled projects only) At a program halt, thread state and status data
Image Viewer (on page 2-124)	A view of BMP, JPEG, PPM, or MPEG data from processor memory or from a file on your PC. You can edit, copy, print, or export image data.

Disassembly Windows

By default, a **Disassembly** window appears when you open a new session. You can open a **Disassembly** window by choosing **View**, **Debug Windows**, and **Disassembly**.

Figure 2-29 and Figure 2-30 show examples of **Disassembly** windows, one with and one without the address bar enabled.

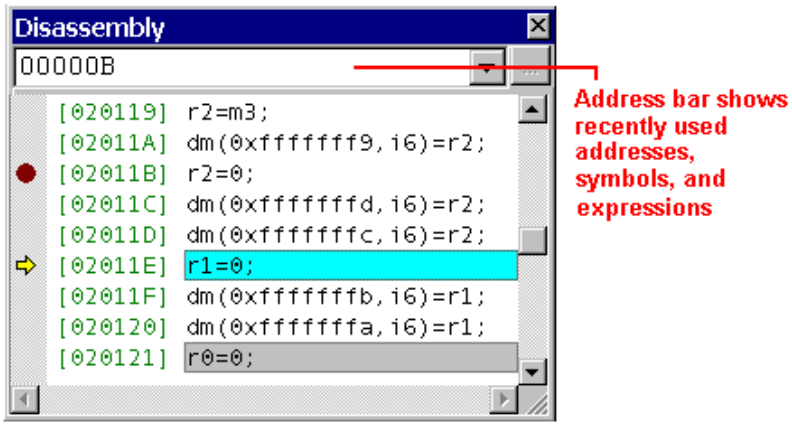

Figure 2-29. Example: Disassembly Window Showing Address Bar

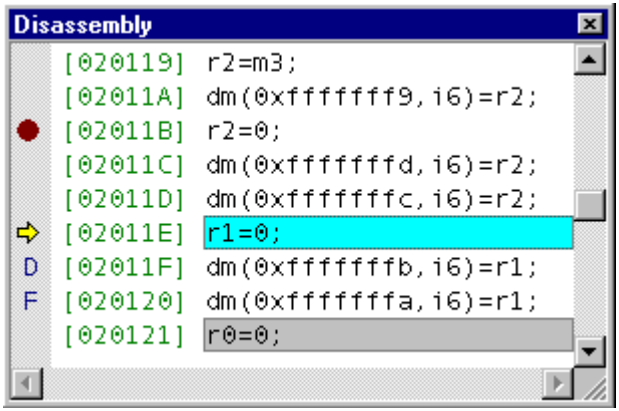

Figure 2-30. Example: Disassembly Window Without Address Bar

Disassembly windows display code in disassembled form, which is useful for temporarily modifying the code to test a change or to view code when no source is available. The **Disassembly** window enables you to examine the assembly code generated by the C/C++ compiler. Choosing **View Source** from the **Disassembly** window's right-click menu enables you to view the C/C++ source code for the loaded file.

A single-processor debug session provides one Disassembly window only, but multiple instances of the window may be opened for different views. In a multiprocessor debug session, multiple Disassembly windows are available.

To make changes permanent, modify the code and rebuild the project.

Disassembly windows provide:

- Number format and edit features, similar to register windows
- Dump and fill capabilities
- Symbols at the far left of the window, denoting program execution stages and pipeline stages

You can enable and disable the display of pipeline symbols in mixed mode (C/C++ and assembly).

- An optional address bar that enables you to navigate to an address, symbol, or expression. The address bar maintains a most recently used history of visited locations.

  To display the address bar, right-click in a **Disassembly** window and choose **Address Bar**. A check mark next to this option on the right-click menu indicates that this feature is enabled.

By default, the current source line to be executed is highlighted by a light-blue horizontal bar, as shown in Figure 2-31.

Figure 2-31. Example: Current Source Line in the Disassembly Window

The color of the current source line and other window items are user configurable. Refer to VisualDSP++ online Help for detail.

Other Disassembly Window Features

From the **Disassembly** window, you can perform the operations described in Table 2-15.

Table 2-15. Disassembly Window Operations

To...	Place the mouse pointer over...
Move to a different address	An address field and double-click. Then select the address from the ensuing **Go To** dialog box. Note that you can also use the address bar to navigate to an address, symbol, or expression.
Insert or remove a breakpoint	An instruction and double-click
Toggle (enable or disable) a breakpoint	An instruction and right-click. Then choose the appropriate command from the ensuing menu.

Right-Click Menu

The **Disassembly** window's right-click menu provides access to the commands shown in Figure 2-32.

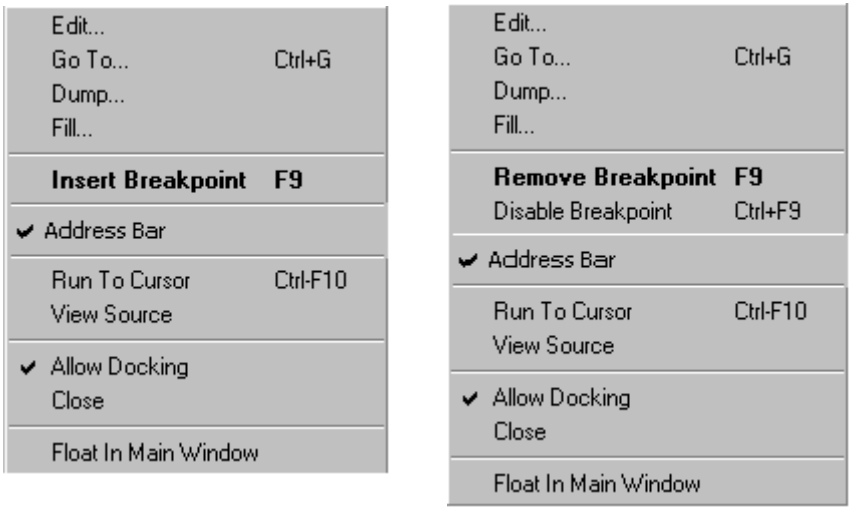

Figure 2-32. Disassembly Window Right-Click Menus

Disassembly Window Symbols

Symbols at the far left of the **Disassembly** window indicate program execution stages (Table 2-16).

Table 2-16. Disassembly Window Symbols

Symbol	Description
	(Gray arrow) The current instruction is being aborted due to a branch or jump instruction.
	A breakpoint is enabled.
	A breakpoint is disabled.
F	This instruction is currently in the Fetch Address stage of the pipeline.
P	(ADSP-2136x targets only) This instruction is currently in the Predecode (Fetch2) stage of the pipeline.
D	This instruction is currently in the Instruction Decode stage of the pipeline.
A	(ADSP-2136x targets only) This instruction is currently in the Address Decode stage of the pipeline.
E	(SHARC targets only) The instruction is in the Execute pipeline stage.
	(Yellow arrow) This instruction is currently in the Execute stage of the pipeline.

The display of pipeline stages is available only when:

- The session is connected to a SHARC simulator target
- **Enable pipeline display** is selected on the **General** tab page of the **Preferences** dialog box, available from the **Settings** menu

Expressions Window

The **Expressions** window (Figure 2-33) lets you enter an expression to evaluate in your program. Evaluations are based on the current debug context. Open this window by choosing **View**, **Debug Windows**, and **Expressions**.

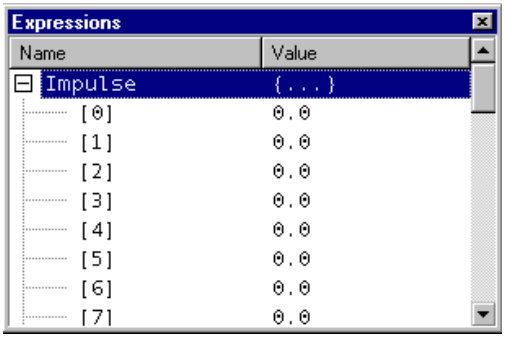

Figure 2-33. Expressions Window

Because of the way registers are saved and restored on the stack, the register value on which the expression relies may be incorrect if you change VisualDSP++'s context from the **Call Stack** window.

Right-Click Menu

The **Expressions** window's right-click menu (Figure 2-34) provides commands that change the display's number format.

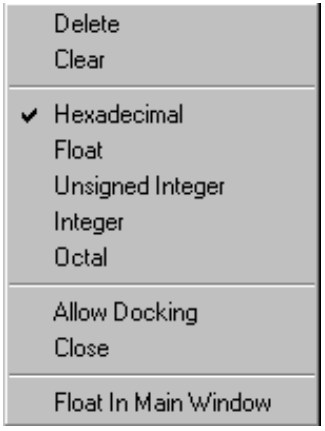

Figure 2-34. Example: Expressions Window Right-Click Menu

Expressions Permitted in an Expression Window

Table 2-17 lists and describes the types of expressions that may be entered in an **Expressions** window.

Table 2-17. Types of Expressions Permitted in an Expressions Window

Expression	Description
Memory address	Precede memory identifiers with a `$` sign, for example: `$dm(0xF0000000)`
Register expression	Precede register names with a `$` sign, for example: `$r0`, `$r1`, `$ipend`, `$po`, or `$imask`
C/C++ statements	Use standard C/C++ arithmetic and logical operators.
Multiprocessor expression (emulator sessions only)	Expressions can be evaluated on a particular processor by using the format: `@processor_name(expression)` where `processor_name` is the name of one of your MP processors, and expression is the expression that you want to evaluate on that processor. For example, on an MP system with two processors (master and slave), this expression evaluates the PC register on master: `@master($PC)`

As you step through your program, the **Expressions** window displays the current value of each listed expression. Expressions evaluation is based on the current debug context.

For example, if you enter expression "`a`" and a global variable "`a`" exists, you see its value. If you then step into a function that has local variable "`a`", the local value displays until the debug context leaves the function. When a variable goes out of context, a string displays next to the variable, informing you that the variable is out of context.

The expressions described above are C expressions. The current syntax also allows the use of registers in expressions.

For example, the following expression is valid.

```
$R0 + $I0
```

Register expressions and C expressions can be mixed in an expression.

Register expressions follow these rules:

- Precede register names with a dollar sign character (`$`).
- Register names can be uppercase or lowercase characters.
- Registers have no context. A register expression always evaluates to the current value of the register.

Trace Windows

(SHARC processors only)
Perform a trace (also called an execution trace or a program trace) to analyze the run-time behavior of a processor program, to enable I/O capabilities, and to simulate source-to-target data streaming. Figure 2-35 shows an example of data in a **Trace** window.

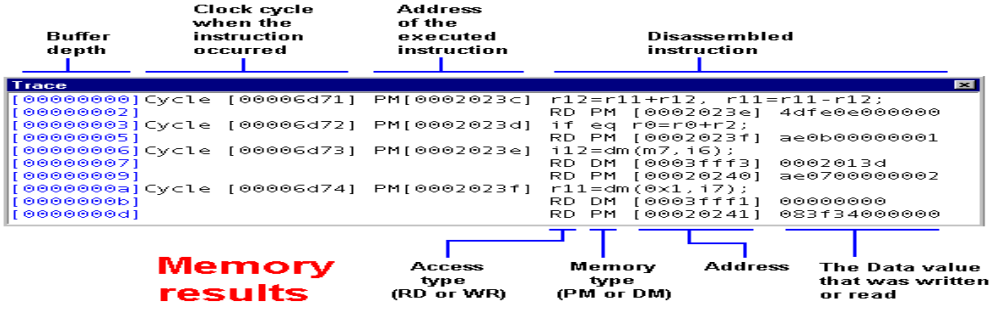
Buffer depth
Clock cycle when the instruction occurred
Address of the executed instruction
Disassembled instruction
Trace
[00000000] Cycle [00006d71] PM[0002023c] r12=r11+r12, r11=r11-r12;
[00000002] RD PM [0002023e] 4dfe0e000000
[00000003] Cycle [00006d72] PM[0002023d] if eq r0=r0+r2;
[00000005] RD PM [0002023f] ae0b00000001
[00000006] Cycle [00006d73] PM[0002023e] i12=dm(m7,16);
[00000007] RD DM [0003fff3] 0002013d
[00000009] RD PM [00020240] ae0700000002
[0000000a] Cycle [00006d74] PM[0002023f] r11=dm(0x1,17);
[0000000b] RD DM [0003fff1] 00000000
[0000000d] RD PM [00020241] 083f34000000
Memory results
Access type (RD or WR)
Memory type (PM or DM)
Address
The Data value that was written or read

Figure 2-35. Example of Data in a Trace Window

The **Trace** window displays:

- Buffer depth (**Custom** in the **Trace Buffer Depth** dialog box)
- The clock cycle when the instruction occurred
- The address of the instruction executed
- The disassembled instruction

Memory results have the following fields.

- Access type (RD or WR)
- Memory type (PM or DM)
- The address, in brackets ([ ])
- The data value written or read

Refer to "Code Analysis Tools" on page 3-7 for related information.

Locals Window

The **Locals** window displays the value of local variables within a function, as shown in Figure 2-36. Open this window from the **View** menu by choosing **Debug Windows** and **Locals**.

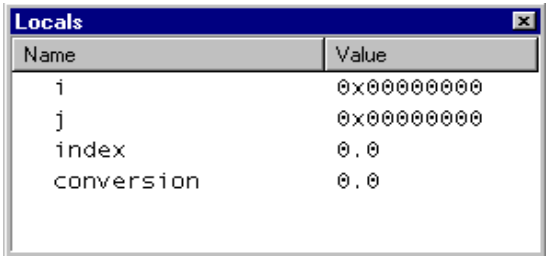

Figure 2-36. Example: Locals Window

Use the **Locals** window with a **Step** or **Halt** command to display the current value of variables when moving through your program.

Complex variables, C structures, and C++ classes appear with a plus sign. Click on the plus sign to display all variable information.

The window's right-click menu provides the commands shown in Figure 2-37.

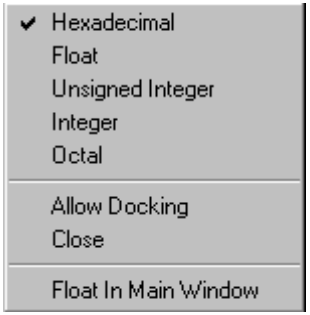

Figure 2-37. Example: Locals Window Right-Click Menu

Statistical/Linear Profiling Results Window

To open a profiling results window, choose **Tools**, (**Statistical Profiling** or **Linear Profiling**), and **New**. Depending on the target, the window's title is **Statistical Profiling Results** or **Linear Profiling Results**. The window comprises two panes, as shown in Figure 2-38.

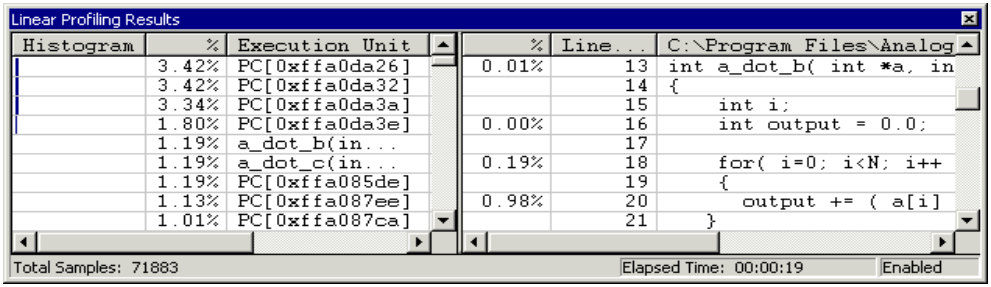

Figure 2-38. Example of a Linear Profiling Results Window

Window Components

The window, which comprises two panes and a status bar, provides a right-click menu for performing various window functions.

Left Pane

The window's left pane displays a list of the executed functions, assembly source lines, and PCs (with no debug information). The time that each item spends on execution appears as a histogram and as a percent. The order of the items in the display is determined by the percentage of global execution time for each item.

The left pane includes the information described in Table 2-18.

Table 2-18. Left Pane Information

Column	Displays	Purpose
Histogram	Horizontal bars	Graphically represents the execution percentage
% -or- Count	A percent with two decimal places, for example: 15.01% -or- a number	Displays execution in percent or as a count. Right-click and choose **View Execution Percent** to view execution as a percent, or choose **View Sample Count** to view the PC sample count.
Execution Unit	Functions, assembly source lines, and PCs for which no debug information exists	These items are sorted by the percentage of global execution time that each item took to execute. The highest percentage items appear at the top of the list

Double-clicking on a line with a function or assembly source line in the left pane displays the corresponding source file in the right pane. The top of the function or assembly source line is shown in the source file. If you double-click on a PC address with no debug information, the **Disassembly** window opens to that address.

Right Pane

The right pane includes the information described in Table 2-19.

Table 2-19. Information in the Right Pane

Column	Displays
%	Execution percent in text format with two decimal places, for example: 1.03% -or- the PC sample count for each source line
Line	Line numbers of the source file
File	Entire source file. Each source line occupies one line in the grid control.

Status Bar

The status bar at the bottom of the window indicates the total number of collected PC samples, the total elapsed time, and indicates whether statistical profiling is enabled.

Right-Click Menu

The **Statistical Profiling Results** and **Linear Profiling Results** windows provide a right-click menu. The menu commands depend on the context (whether you right-click in the left pane or right pane) and the current settings.

Table 2-20 describes the right-click menu commands.

Table 2-20. Profiling Results Window Right-Click Menu Commands

Command	Description
Enable	Enables or disables profiling
Load Profile	Opens the **Select a Statistical /Linear Profile to Load** dialog box from which you can load profile data saved from a previous run

Table 2-20. Profiling Results Window Right-Click Menu Commands

Command	Description
Save Profile	Saves the current run's data to a file
Concatenate Profile	Merges profiling data stored from a previous run with current data
Clear Profile	Clears statistics saved from a previous run
View Execution Percent	Displays the execution percent in each execution unit or source line. This value is the sample count for that execution unit divided by the total number of samples.
View Sample Count	Displays the sample count for that execution unit
Mixed -or- Source	Sets the display mode for C/C++ source lines from the right pane only. Choose **Mixed** to display both C/C++ source lines and assembly lines. C/C++ source lines appear in black type, and assembly lines appear in gray. Profiling data appears for each assembly line. Choose **Source** to display only the C/C++ source lines.
Properties	Opens the **Profile Window Properties** dialog box, where you can view or change window settings. When performing linear profiling, you can select display options such as cache hits, cache misses, execution count, reads, and writes.

Window Operations

You can select various options for the **Statistical/Linear Profiling Results** window and perform various window operations.

Changing the Window View

After specifying window properties for the **Statistical/Linear Profiling Results** and enabling profiling, the profiler collects data when you run a program. Depending on the filtering options selected, the window's **Execution Unit** column displays:

- Function names (such as `main`)
- Single addresses, for example, `PC(0x2000)`
- Address ranges, for example, [`2000-2050`]

Single addresses and address ranges display in hexadecimal format. The "0x" notation, however, appears beside single addresses only.

Displaying a Source File

Double-clicking on a function name in the **Execution Unit** column not only displays the source of the function in the right pane but also displays profiling data for each line of the function. Figure 2-39 shows an example of code displayed for a function.

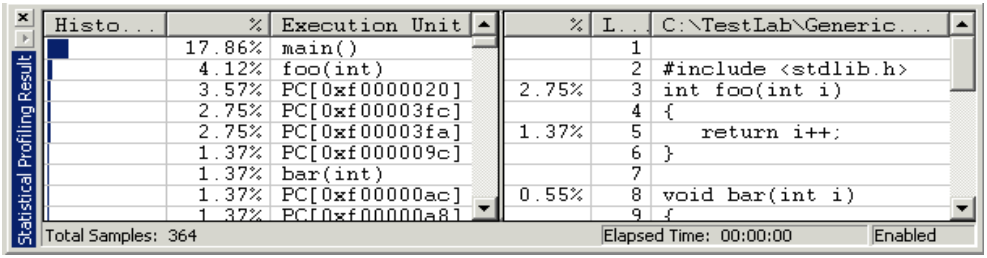

Figure 2-39. Example: Code Displayed for a Function

Displaying Functions in Libraries

The profiling window enables you to display functions in libraries, as shown in Figure 2-40.

To use this feature, right-click in the profiling window and choose **Properties** to open the **Profile Window Properties** dialog box.

Next, click the **Filter** tab, select **C/C++ functions**, and click the **Add** button to open the **Add Functions** dialog box. Then select the **Show all functions** option.

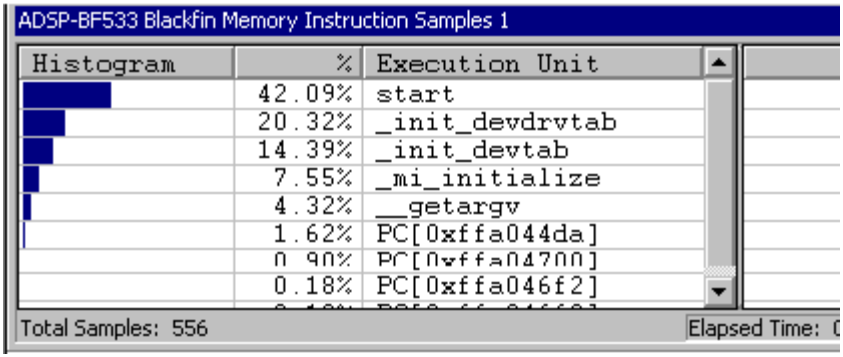

Figure 2-40. Profiling Window Showing Library Functions

Working With Ranges

Clicking on the icon in an address range expands or contracts the list of functions within that address range.

When expanded, the list of functions appears and the profiling data appear immediately after the address range.

Switching Display Modes

The right-click menu's **Mixed** and **Source** commands simplify switching between two views. Figure 2-41 shows the source mode view and Figure 2-42 shows the mixed mode view.

You can display the right pane in source mode

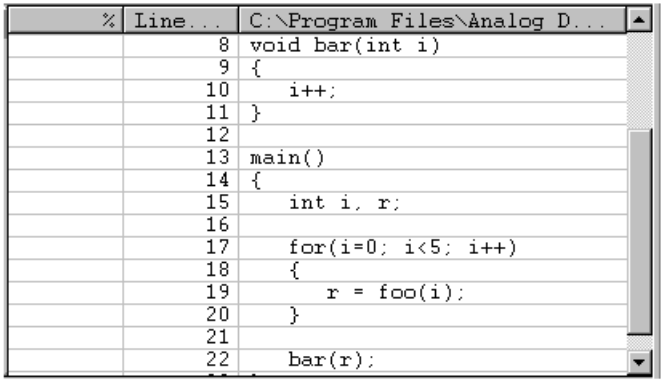

Figure 2-41. Source Mode View

Mixed mode dispalys C/C++ source lines and assembly code.

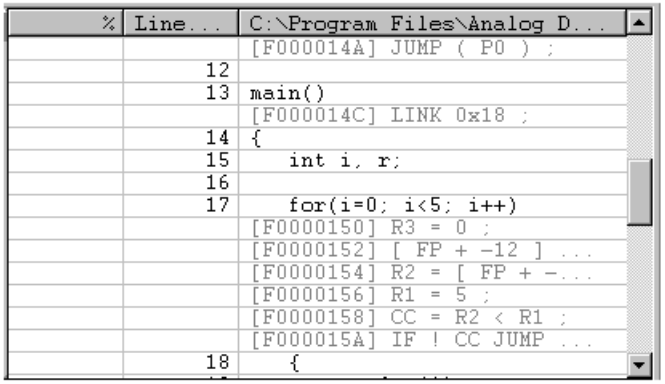

Figure 2-42. Mixed Mode View

When you view the window in mixed mode, profiling data for each assembly line displays, as shown in Figure 2-43. Mixed mode displays profiling statistics for individual assembly instructions.

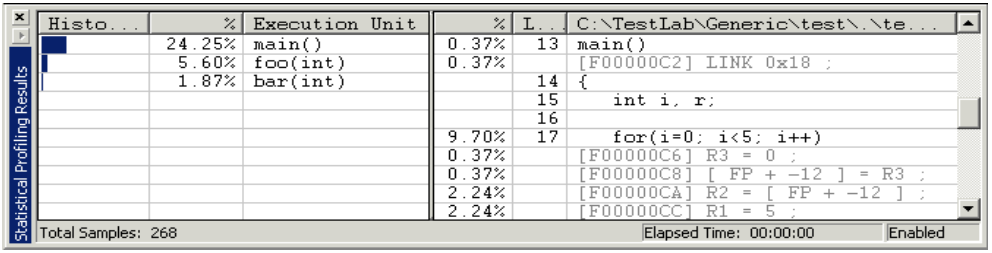

Figure 2-43. Profiling Data for Each Assembly Line (Mixed Mode)

Filtering PC Samples With No Debug Information

Since you spend most of your time building a “debug version” of your code, eliminate non-debug code, such as C run-time library initialization code.

The profiling results in Figure 2-44 show where a lot of time is spent before filtering.

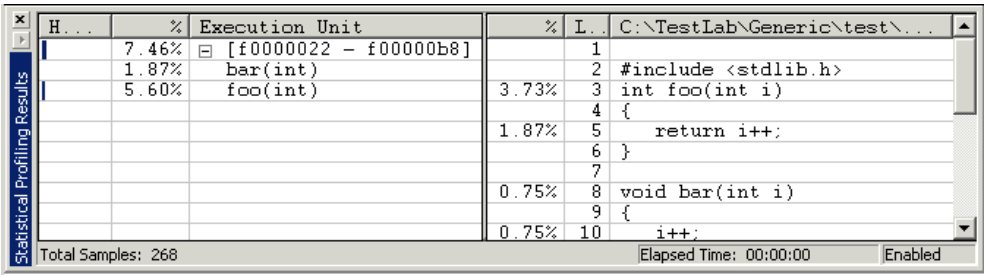

Figure 2-44. Profiling Results Before Filtering

The profiling results after filtering (Figure 2-45) reflect the difference.

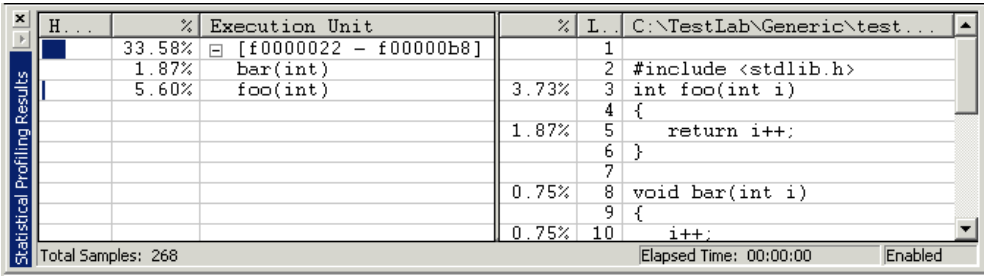

Figure 2-45. Profiling Results After Filtering

Call Stack Window

The **Call Stack** window (Figure 2-46) enables you to double-click on a stack location to move the call stack back to a previous debug context. Open this window by choosing **View**, **Debug Windows**, and **Call Stack**.

Figure 2-46. Call Stack Window

The **Call Stack** window functions with C/C++ code only. Use this window to analyze the state of parent functions when erroneous data is being passed to the currently executing function and to see the context from which the current function is being called. You can walk up the call stack and view local variables in different scopes.

Memory Windows

Use Memory windows to:

- View and edit memory contents
- Display the address of a value. Move the mouse pointer over the value, and hold down the keyboard's **Ctrl** key.
- Lock the number of columns currently displayed. This action resizes the window horizontally without altering the display.
- Track one expression

You open memory windows from the **Memory** menu.

Memory windows provide:

- Number format and edit features
- Fill and dump capabilities
- An optional address bar for fast navigation to recently used addresses, symbols, or expressions

  To display the address bar, right-click in a memory window and choose **Address Bar**. A check mark next to this command on the right-click menu indicates that this option is enabled.

Number Formats in Memory Windows

The memory windows in Figures 2-53, 2-54, 2-55, and 2-56 show examples of different memory number formats.

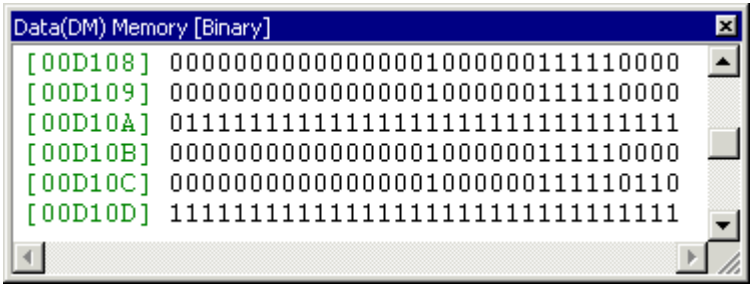

Figure 2-47. SHARC Memory Window in Binary Format

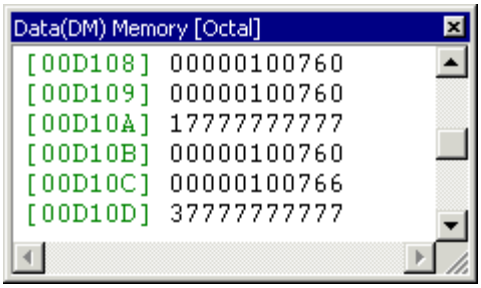

Figure 2-48. SHARC Memory Window in Octal Format

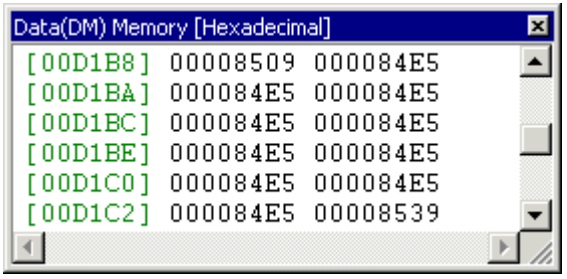

Figure 2-49. SHARC Memory Window in Hexadecimal Format

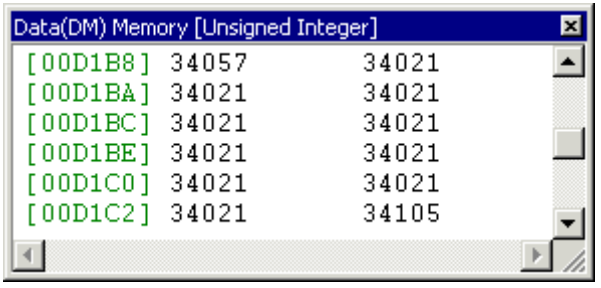

Figure 2-50. SHARC Memory Window in Unsigned Integer Format

Right-Click Menu

Memory windows provide a right-click menu. Choosing the **Select Format** command enables you to change the display's number format. Figure 2-51 shows the formats available for SHARC processors.

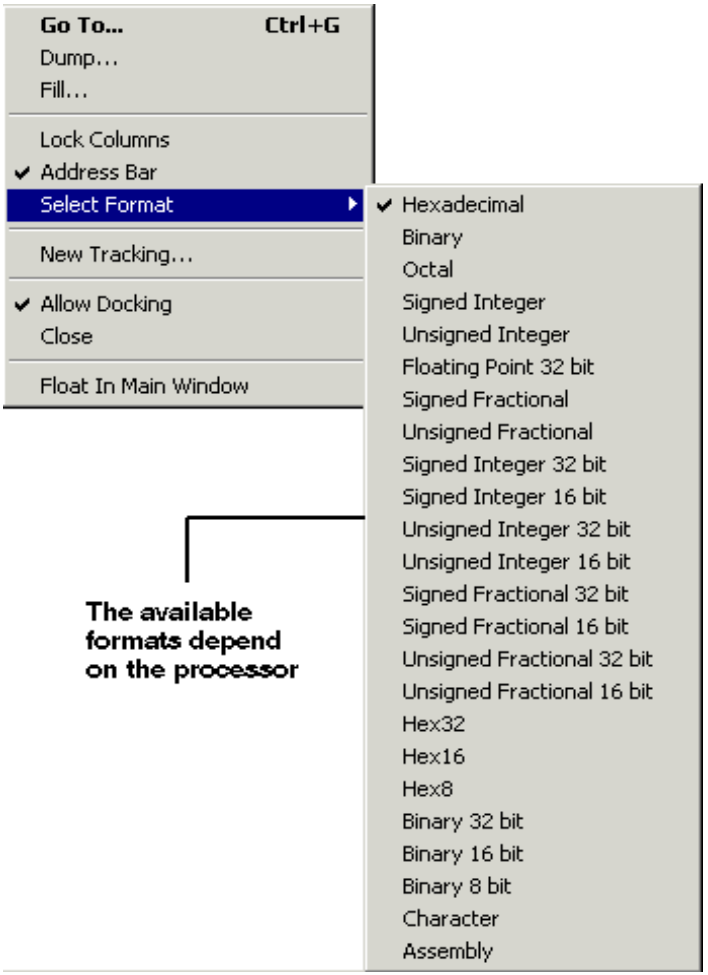

Figure 2-51. Example: Memory Window Formats for SHARC Processors

Expression Tracking in a Memory Window

While stepping through code, a memory window configured for expression tracking shows the memory at the address specified by the expression.

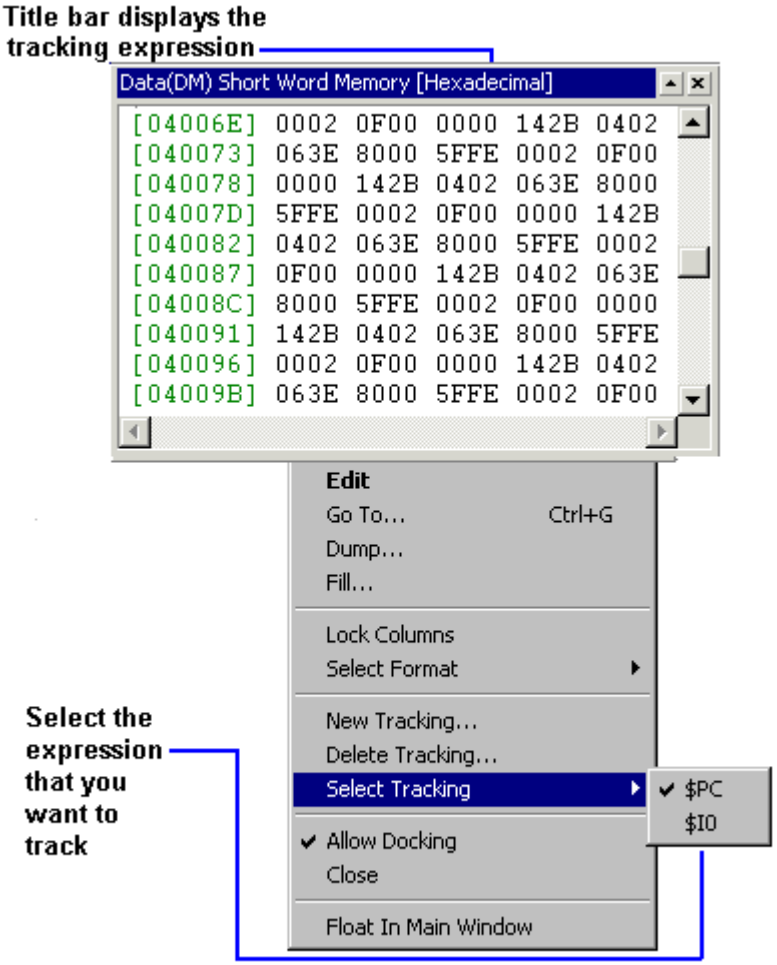

Figure 2-52. Expression Tracking in a Memory Window

When the target halts, the tracking expression is evaluated and the memory window jumps to that address. For example, when the tracking expression is "`$PC`", the memory window behaves like the **Disassembly** window.

Rules

- In a memory window, several expressions for tracking can be configured.
- In a memory window, only one expression (the active expression) can be tracked at any time.
- The active expression appears in the memory window's title bar.
- The memory window's right-click menu displays a list of configured expressions, from which to select only one expression for tracking.
- To track multiple expressions, open multiple memory windows and track one expression per window.

Memory Window Display Customization

You can specify the colors used for symbols, data, address values modified values, and undefined memory regions. You can also adjust the width of the window to display a particular number of data columns. For example, the memory window in Figure 2-53 is sized to display five columns.

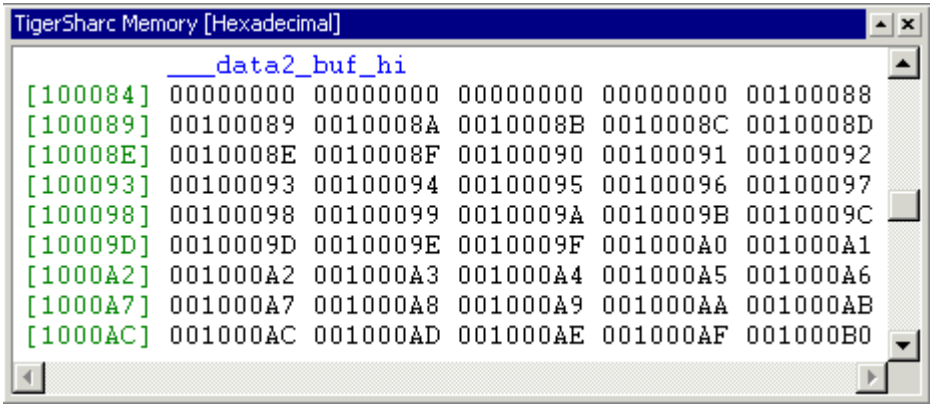

Figure 2-53. Example: Memory Window Sized to Display Five Columns

The commands available via the menu bar's **Memory** menu and the memory window's right-click menu depend on the processor that you are debugging.

Background Telemetry Channels (BTCs)

Background telemetry channels (BTCs) enable VisualDSP++ and a processor to exchange data via the JTAG interface while the processor is executing. Before BTC, all communication between VisualDSP++ and a processor took place while the processor was in a halted state.

Background telemetry channels are supported only in SHARC and Blackfin emulator sessions. For information about using BTCs, refer to the *VisualDSP++ 4.0 Getting Started Guide* and online Help.

BTC Definitions in Your Program

Background telemetry channels are defined on a per program (.DXE) basis. The channels are defined when a specific program is loaded onto a processor. Define channels in your program by using simple macros.

The following example code shows channel definitions.

```
#include "btc.h"

.section/DM seg_dmda;   // for ADSP-2126x processors

BTC_MAP_BEGIN
   BTC_MAP_ENTRY ('Channel0', 0xf0001000, 0x00100)
   BTC_MAP_ENTRY ('Channel1', 0xf0002000, 0x01000)
   BTC_MAP_ENTRY ('Channel2', 0xf0003000, 0x10000)
BTC_MAP_END
```

The first step in defining channels in a program is to include the BTC macros by using the `#include btc.h` statement. Then each channel is defined with the macros. The definitions begin with `BTC_MAP_BEGIN`, which marks the beginning of the BTC map. Next, each individual channel is defined with the `BTC_MAP_ENTRY` macro, which takes the parameters described in Table 2-21.

Table 2-21. Parameters for the BTC_MAP_ENTRY_ASM Macro

Parameter	Description
Name	Name of the channel (32 characters max)
Starting address	Starting address of the channel in memory
Length	Length of the channel in 32-bit words for ADSP-2126x processors

Once the channels are defined, end the BTC map by using the `BTC_MAP_END` macro.

Enabling BTC on ADSP-2126x and ADSP-BF36x Processors

After the channel definitions are added, the BTC must be initialized with a call to the `_btc_init` function during the application's start-up code.

After initialization, BTC commands from the host are processed via the low-priority emulator interrupt (`EMULI`). A vector to the interrupt service routine must first be installed.

In assembly language, the vector can be installed with a jump to a `_btc_isr` instruction placed at the `EMULI` vector location:

```
JUMP _btc_isr;
```

After the interrupt vector is installed, the interrupt itself must be enabled with the following code:

```
// setup imask
ustat1 = imask;
BIT SET ustat1 EMULI;
imask = ustat1;

// enable interrupts
ustat1 =  mode1;
BIT SET ustat1 IRPTEN;
mode1 = ustat1;
```

In C/C++, the vector can be installed with the `interrupt` function as follows:

```
interrupt (SIG_EMUL, btc_isr);
```

In C/C++, the interrupt function enables the interrupt for you.

After adding code to initialize BTC and enable the BTC interrupt, you must link with the BTC library (`libbtc26x.dlb` for assembly applications or `libcbtc26x.dlb` for C/C++ applications). This library contains the initialization function, interrupt service routine, and other functions that permit data transfer over the BTC.

BTC Priority

On ADSP-2126x and ADSP-BFBF36x SHARC processors, BTC data transfer is handled through the low-priority emulator interrupt (`EMULI`). Since the priority of this interrupt is fixed, the priority of BTC is also fixed.

The priority of the BTC can impact the response time from when the host requests data and the processor responds. Once the processor begins to service the request, interrupts can still be serviced by the processor. BTC performance can be affected by the frequency of system interrupts.

BTC Memory Window

The **BTC Memory** window lets you view background telemetry channel contents in real time. The window displays the contents of the address that you want to see. Change the window's view to meet your needs.

Open this window by choosing **View**, **Debug Windows**, and **BTC Memory**.

The view in Figure 2-54 shows the contents of a specified channel only (for example, Channel1).

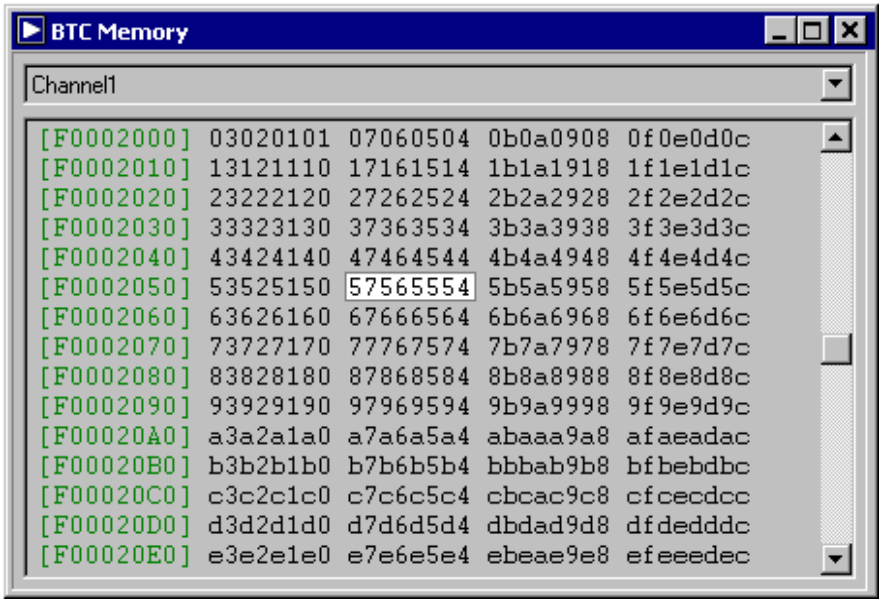

Figure 2-54. Example: Viewing Contents of a Specified Channel Only

The view in Figure 2-55 shows the list of currently defined channels and the contents of the selected channel.

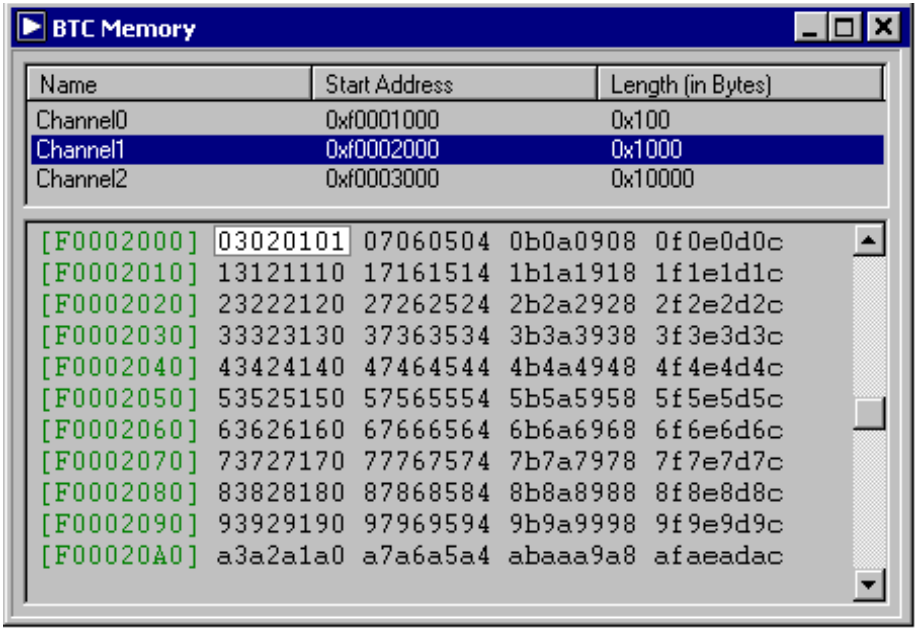

Figure 2-55. Viewing Defined Channels and Contents of a Selected Channel

Right-Click Menu

Table 2-22 describes the **BTC Memory** window's right-click menu.

Table 2-22. BTC Memory Window Right-Click Menu

Command	Purpose
Go To	Opens the **Go To** dialog box, in which you specify an address. The specified address appears in the top-left corner of the display. The address must be within the range defined for the channel currently being displayed. **Tip:** Double-clicking in the address column also opens the **Go To** dialog box.
Show Map or Hide Map	Shows or hides a more informative map display of all the current channel definitions **Show Map** displays a channel list. Double-click a channel to display its contents in the lower portion of the window. **Hide Map** removes the list of channels. The selected channel remains in the display.
Lock Columns	Locks or unlocks the number of columns currently displayed in the window
Select Format	Specifies how to display data in the window. Choices include double words (32 bits), words (16 bits), and bytes (8 bits).
Refresh Rate	Specifies the refresh rate, which is used when **Auto Refresh** is chosen. The display is updated at the selected interval.
Auto Refresh	Enables the window to refresh itself at given intervals. The rate is specified by **Refresh Rate**. **Auto Refresh** mode is valid only while the processor is running.
Channel Timeout	Specifies the length of time to wait for any single response from the BTC. If the timeout value is exceeded, the current transaction ends.

Memory Map Windows

A **Memory Map** window (Figure 2-56) displays the memory map for the selected processor. Open this window by choosing **Memory** and **Memory Map**.

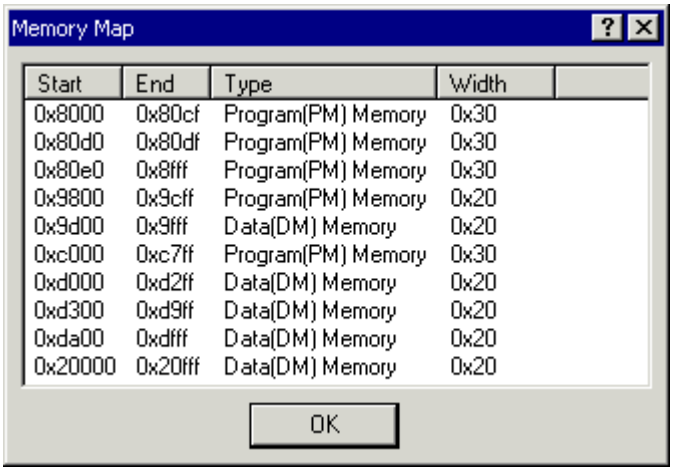

Figure 2-56. Example: Memory Map Window

If no program is loaded into the processor, the memory map displays all available memory in the processor.

If a program is loaded, the memory map is the map defined in the `MEMORY` section of the program's `.LDF` file.

For each portion of memory, the window displays the start address, end address, and width.

Register Windows

Access various register windows via the VisualDSP++ **Register** menu. The available commands (and subsequent windows) depend on the processor.

Figure 2-57 shows an example **Register** menu tree for a SHARC processor.

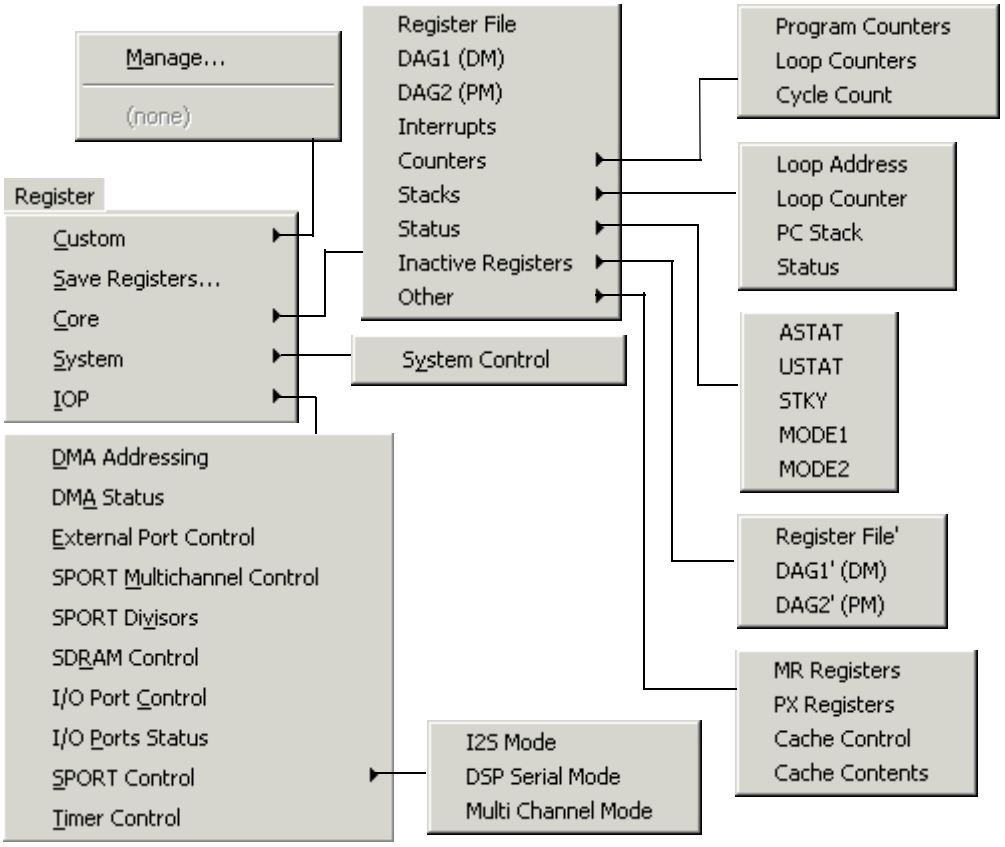

Figure 2-57. Example: Register Windows for a SHARC Processors

Figure 2-58 shows an example **Register** menu tree for a TigerSHARC processor.

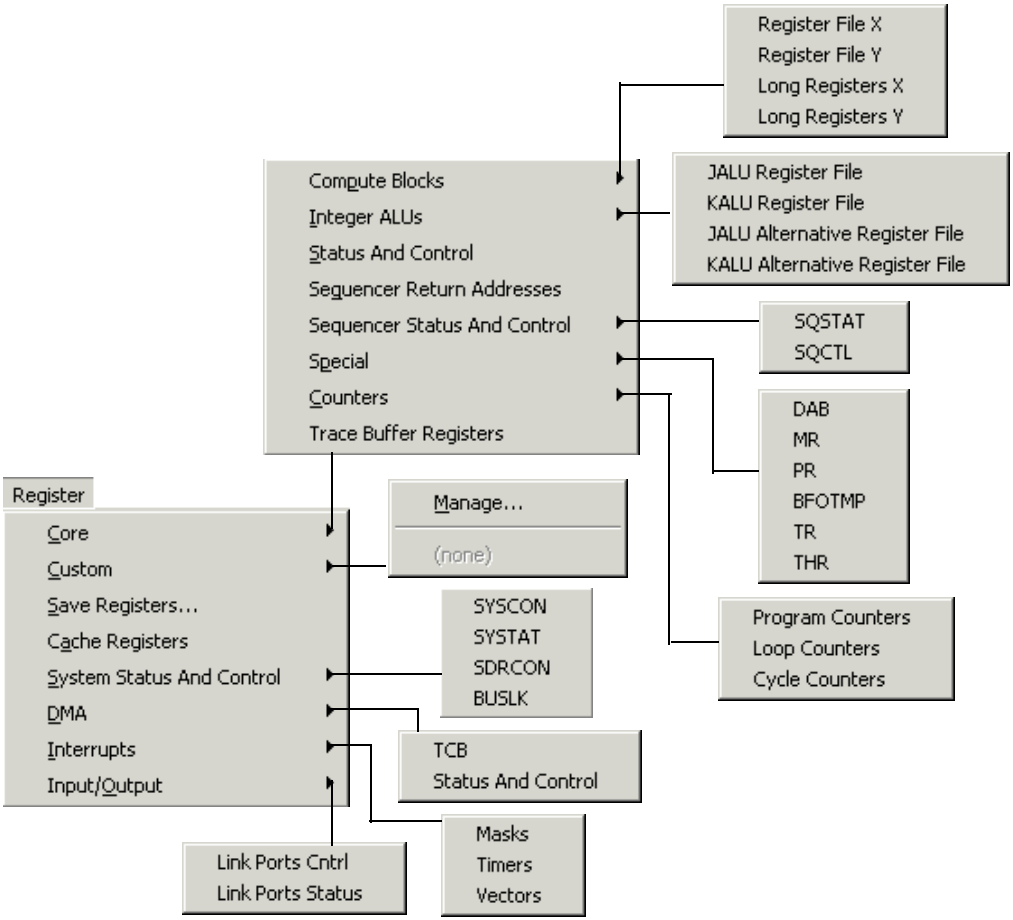

Figure 2-58. Example: Register Windows for a TigerSHARC Processor

Figure 2-59 shows an example of a data register file in a register window.

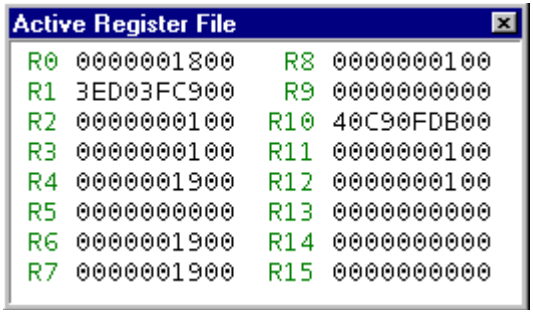

Figure 2-59. Example Register Window

A register window enables you to:

- View and change register contents
- Change the window's presentation (number format)

Register window number formats include standard formats, such as hexadecimal, octal, and binary. Depending on the processor, other formats may be available.

You can change a register's data directly from within a register window. The modified register content is used during program execution. Edits to data do not affect your source files. To make changes permanent, edit the source file and rebuild your project.

Stack Windows

Depending on your processor, access to various stack windows is available, including:

- PC Stack
- Counter Stack
- Loop Stack
- Status Stack

Access stack windows via the **Register** menu. For more information about your processor's stack windows, consult VisualDSP++ online Help.

Custom Registers Windows

While debugging, you can configure and display **Custom Registers** windows. To create a **Custom Registers** window, choose **Register**, **Custom**, and **Manage**. Then configure and add the registers that you want to display. The **Custom Registers** window appears immediately after it is created.

Each **Custom Registers** window displays a customizable title and the registers that you choose to monitor. The **Custom Registers** window shown in Figure 2-60 displays the contents of five registers.

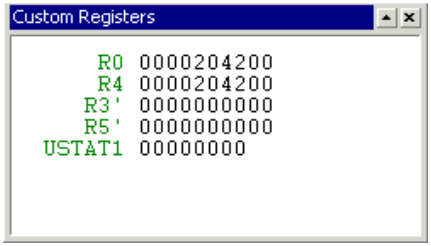

Figure 2-60. Example: A Customized Registers Window

Multiprocessor Window

The SHARC and TigerSHARC simulators do not support multiprocessor (MP) debugging; multiprocessing for these processors is available in emulator sessions only. Multiprocessing support for Blackfin ADSP-BF561 and ADSP-BF566 processors is available in simulation.

Use the **Multiprocessor** window (Figure 2-61) to select and control the different processors in a multiprocessor debug session.

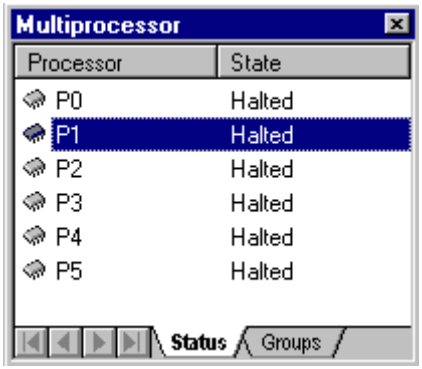

Figure 2-61. Example: Multiprocessor Window

Multiprocessor Window Pages

The **Multiprocessor** window has two tabbed pages, **Status** and **Groups**.

Status Page

The **Status** page (Figure 2-62) shows the status of each processor in the multiprocessor system. A horizontal bar highlights the processor with focus.

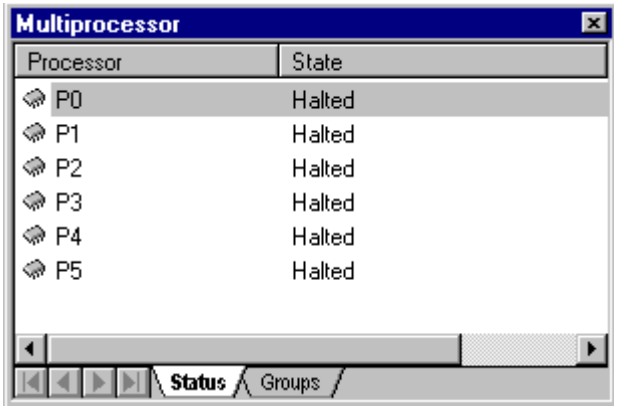

Figure 2-62. Multiprocessor Window – Status Page

Change focus by clicking on a processor in the list.

Groups Page

The **Groups** page (Figure 2-63) shows the current list of multiprocessor groups. A **Default** group is created with each new multiprocessor session. The members of the **Default** group are the processors that you checked off under **Multiprocessor System** in the **New Session** dialog box.

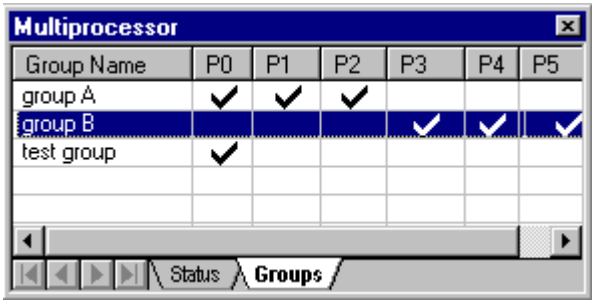

Figure 2-63. Example: Multiprocessor Window – Groups Page

From the **Groups** page, assign one or more processors to a group. Performing a multiprocessor operation (**MP Run**, **MP Halt**, **MP Step**, **MP Reset**, and **MP Restart**) affects only the processors in the currently selected group.

Right-clicking on the **Group** page displays a context menu for adding or removing a group.

Operating on Multiprocessor Groups

For example, if a session contains three processors (A, B, and C) and a group is created that contains A and C. Running the **MP Run** command runs A and C only, and B remains unaffected.

Focus

Processor focus changes, depending on the window currently selected. To move focus among the processors, click on a processor listed in the **Multiprocessor** window (Figure 2-61).

You can pin a register window, a memory window, or **Disassembly** window to a specific processor. Select the processor in the **Multiprocessor** window and right-click in the window that you want to pin. Then choose **Pin to Processor** to lock the window to the selected processor. A window pinned to a processor always displays that processor's data, regardless of the currently focused processor.

For example, if a register window is pinned to Processor 1 and a memory window is pinned to Processor 2, selecting the register window moves the focus to Processor 1. Selecting the memory window moves the focus to Processor 2. The **Multiprocessor** window's **Status** page reflects the change in focus.

Right-Click Menu

The **Multiprocessor** window's right-click menu offers these commands:

Figure 2-64. Multiprocessor Window's Right-Click Menu

Pipeline Viewer Window

(TigerSHARC and Blackfin processors only in simulation) The **Pipeline Viewer** window (Figure 2-65) displays instructions in the pipeline and allows you to view event details. Open this window by choosing **View**, **Debug Windows**, and **Pipeline Viewer**.

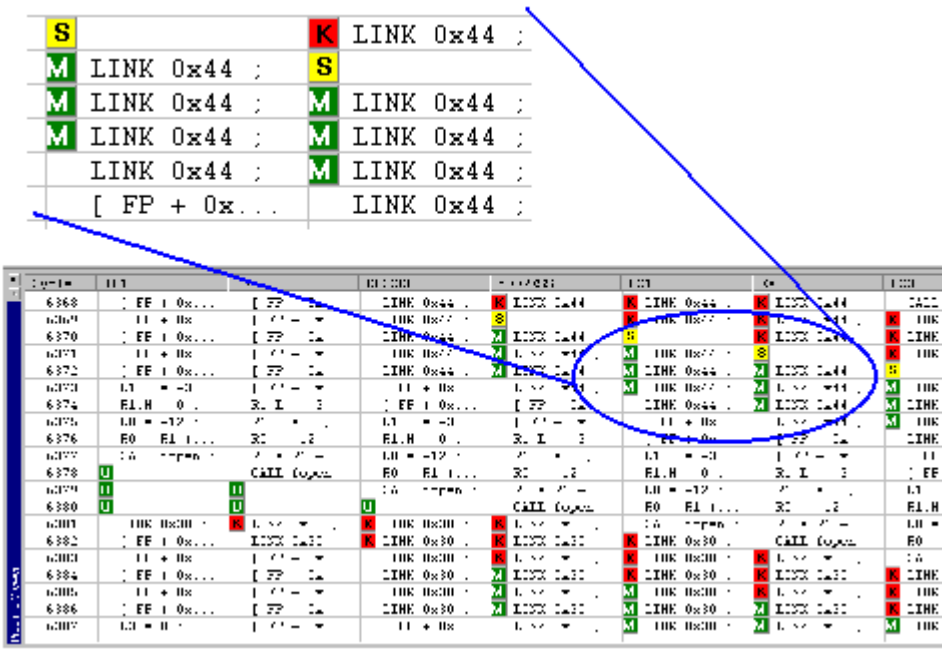

Figure 2-65. Pipeline Viewer Window

For SHARC processors, the **Disassembly** window displays symbols (F, D, or E), indicating an instruction's pipeline stage.

Column headings refer to pipeline stages for the processor's core registers. Refer to your processor's hardware documentation for details.

Right-Click Menu

The **Pipeline Viewer** window's right-click menu provides the commands described in Table 2-23.

Table 2-23. Pipeline Viewer Right-Click Menu

Item	Purpose
Enabled	Enables and disables collection of pipeline data while running or stepping
Clear	Empties the current sample buffer
Display Format	Controls the display format of data **Address** shows the hexadecimal-formatted address of the pipeline stage (for example, `0x1234`). Use this format to follow a particular address route through the pipeline. **Disassembly** disassembles the instruction at that address and shows the opcode mnemonic, similar to a **Disassembly** window. Use this format to determine why a particular event is occurring. **Opcode** format is the hexadecimal representation of the disassembly mnemonic.
Save	Opens the **Save As** dialog box, where you export the collected data to a text file
Properties	Opens the **Pipeline Viewer Properties** dialog box, where you view and specify properties (buffer and display depth, display format, column widths, grid lines, and the appearance of stages) for the **Pipeline Viewer** window. You can also modify window colors.

Pipeline Viewer Properties Dialog Box

From the **Pipeline Viewer Properties** dialog box, you can specify how (depth and format) the **Pipeline Viewer** window displays pipeline events. Table 2-24 describes the Pipeline Viewer properties.

Table 2-24. Pipeline Viewer Properties

Property Item	Purpose
Buffer depth	Specifies the total number of pipeline samples to retain at any time. When this buffer overflows, the oldest data shifts out to make room for new samples. The default is `100`.
Display depth	Specifies the number of samples to display. Adjust this number to meet your performance needs. The lower the depth, the faster the target can run. This option cannot be set greater than the **Buffer depth**. The default is `20`.
Display format	Specifies the data's format **Address** includes the hexadecimal-formatted address of the pipeline stage (for example, `0x1234`). **Disassembly** includes the opcode mnemonic, similar to the format displayed in a **Disassembly** window. **Opcode** format is the hexadecimal representation of the disassembly mnemonic.
Show gridlines	Toggles the display of gridlines in the window. The default is **On**.
Auto-size columns	Automatically sizes all columns to have the same width as samples are collected. The default is **On**.
Stages to view	Specifies the stages to appear in the window. Note that all stages are collected, but you view only the stages that you select to appear.

From the dialog box's **Colors** tab, you can specify the colors that display in the **Pipeline Viewer** window. The current color appears under **Current Color**. Click a color in the color palette or click **Other** to specify a custom color. Click the **Reset** button to restore the default colors.

Pipeline Viewer Window Event Icons

Table 2-25 shows **Pipeline Viewer** window icons that indicate pipeline stage events for ADSP-TS101 and ADSP-TS20x TigerSHARC processors.

Table 2-25. Pipeline Viewer Event Icons

Icon	Event	Description
	Abort	A stage contains an instruction that has been aborted.
	Invalid Instruction in fetch pipe	A stage contains a placeholder representing a result of an invalid fetch. This condition occurs when an instruction alignment buffer is full; or the fetch pipe was flushed because of an abort in the execution pipeline.
	Stall	A stall has been generated at a stage of the pipeline.
	Wait	An instruction at a stage of the pipeline waits to be executed (because of a stall down the pipeline).
	Bubble	The pipeline stage contains an invalid instruction as a result of a stall up the pipeline.
	Hit	An instruction at a stage is a Branch Target Buffer Hit. The address of the last slot of the instruction line was found in the Branch Target Buffer.

The icons in the above table are listed in descending priority. When more than one event occurs at a certain stage at a certain cycle, only one icon displays—the icon with highest priority. For example, if an instruction that was a Branch Target Buffer hit is aborted, the **Abort** icon appears.

Pipeline Instruction Event Details

To view event details, press and hold down the keyboad's **Ctrl** key and move the mouse pointer over the cell in the **Pipeline Viewer** window. The pipeline event details appear in a tool tip (message) box, as shown in Figure 2-66.

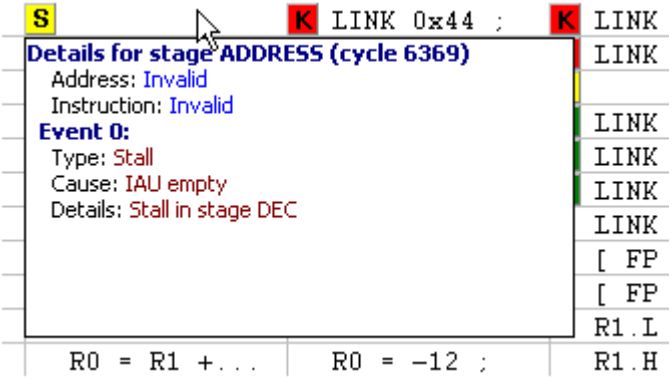

Figure 2-66. Example: Tool Tip Box Showing Pipeline Event Details

A pipeline event can include the details described in Table 2-26.

Table 2-26. Pipeline Event Details

Item	Displays
Address	Address of the pipeline stage at that cycle (if valid)
Instruction	Assembly instruction of that address (if valid)
Type	Type of event
Cause	Cause of the event condition
Details	Further explanation of the cause of the event (if applicable)

Cache Viewer Window

The VisualDSP++ **Cache Viewer** window provides a means to visualize a processor's cache and locate problem areas. The tool shows how instructions are being executed. Use this valuable information to boost your application's performance.

The **Cache Viewer** window (Figure 2-67) displays each instruction's execution characteristics. Cache Viewer information indicates the type of cache event and describes the cause of the event. Each instruction that executes from cache is marked with an **H** (hit) or an **M** (miss). Hits represent cache instructions executed without a stall. Misses identify instructions fetched from slower parts of memory, because they were not found in cache.

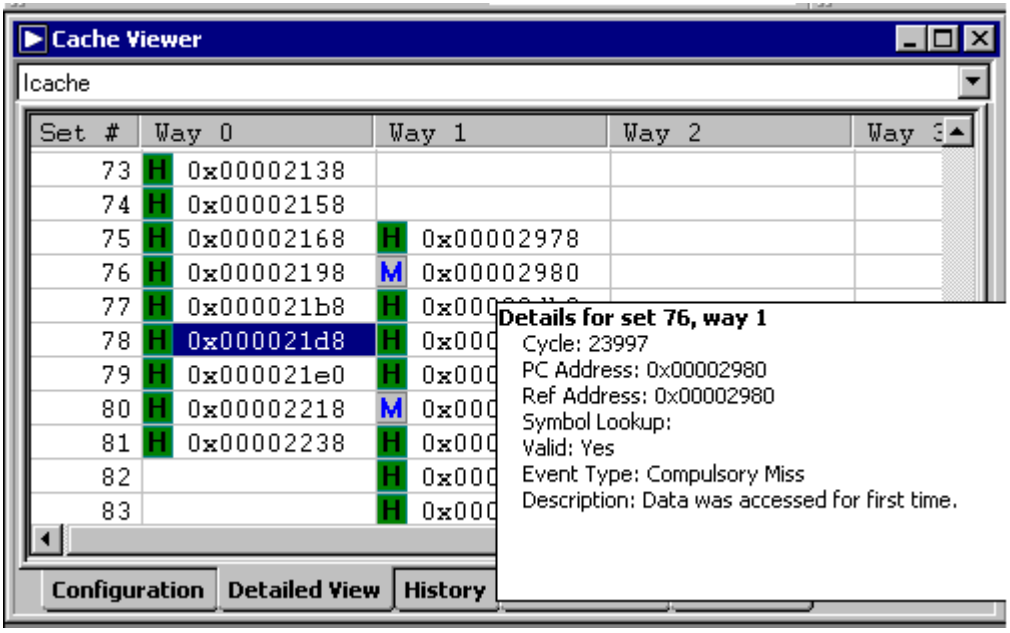

Figure 2-67. Viewing a Cache Event's Details in the Cache Viewer

Use the hit or miss information to increase an application's performance by locating instructions in the cache when they are needed. Ensuring that no cache misses are located in frequently executed areas of an application (as highlighted by the profiler utility) is a critical step in optimizing your application's software performance.

As shown in Figure 2-67, the **Cache Viewer** window enables you to view the details of any cache event. These descriptive details help you understand the cause of the cache event. Use this information to isolate areas where performance can be improved.

For example, based on cache event details, you might:

- Modify an application's layout in memory to avoid cache thrashing
- Prefetch instructions to avoid compulsory misses
- Lock down ways in the cache to avoid a conflict miss with a frequently accessed instruction

Open the **Cache Viewer** window by choosing **View**, **Debug Windows**, and **Cache Viewer**.

The Cache Viewer consists of several tabbed pages, described in Table 2-27.

Table 2-27. Cache Viewer Window Pages

Page	Displays
Configuration	Cache configuration information
Detailed View	Location (set and way) of cache event
History	List of cache events
Performance	Cache performance metrics
Histogram	A plot of cache activity
Address View	Cache events on an Address vs. Cycle plot

The **Cache Viewer** window's right-click menu (described in Table 2-28) enables you to read, write, and step a *cache events log*, which is a file that records cache events.

Table 2-28. Cache Viewer Window's Right-Click Menu

Menu Option	Description
Enabled	Enables and disables collection of cache data while the target is running or stepping
Clear	Clears all displays and deletes all stored cache data
Map References	Opens the **Map References** dialog box, where you specify the cache reference map (start address and end address)
Event Log→Read	Opens the **File Open** dialog box, where you select and open a cache events log file. The log file data is used by the **Cache Viewer** window's **Configuration** view.
Event Log→Write	Opens the **File Save** dialog box, where you save a cache events log file. Cache events are written to this log file.
Event Log→Step	Executes one cache event at a time from the cache events log file. The cache event displays in the **Detailed View**, **History**, and **Histogram** pages of the **Cache Viewer** window. By default, this option is enabled when a cache log file is opened for reading.
Properties	Opens the **Cache Viewer Properties** dialog box, where you specify the **Cache Viewer** window's appearance

The cache events log file does not include icons. Thus, the **Cache Viewer** window's **Detailed View** page does not display icons.

Stepping enables you to execute one cache event at a time from the cache events log file. The cache event displays on the **Detailed View**, **History**, and **Histogram** pages. When stepping is configured, a check mark appears next to the **Step** command on the right-click menu. By default, this option is enabled when a cache events log file is opened for reading.

Configuration Page

The **Configuration** page (Figure 2-68) displays configuration information for configured cache.

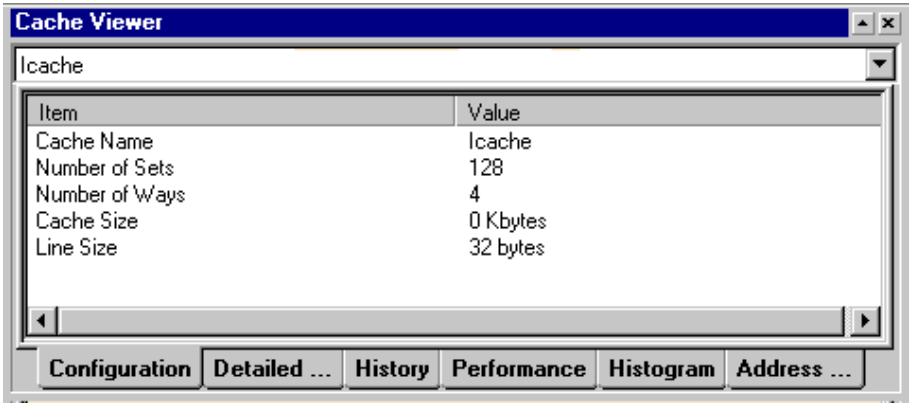

Figure 2-68. Example: Configuration Page

The **Cache Selection** pull-down box (top of dialog box) lists cache displays. When multiple caches are configured, use this list to change cache displays.

The **Cache Configuration** list box (below the **Cache Selection** pull-down box) displays a list of items and their values. The first three items (Cache Name, Number of Sets, and Number of Ways) are required. The target may display additional items, such as Cache Size and Line Size. The list of items depends on the selection in the **Cache Selection** pull-down box.

Detailed View Page

The **Detailed View** page (Figure 2-69) displays a grid, depicting cache sets (rows) and cache ways (columns).

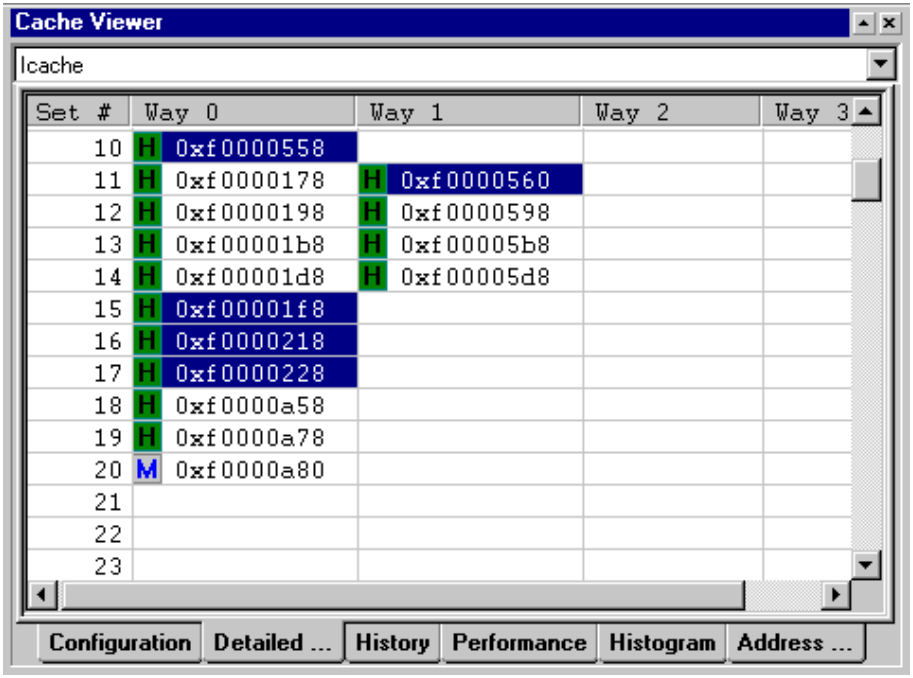

Figure 2-69. Example: Detailed View Page

Data received from a cache event is placed in the cell corresponding to the cache set and way. The most recent cache events are highlighted.

Each cell has an icon and text entry. The icon indicates the type of cache event (hit, miss, and so on) that occurred. Depending on the selected objects, details (such as reference address, PC address, cycle count, event type, event description, and so on) can be shown.

Pressing down the keyboard's **Ctrl** key and moving the mouse over a cell displays a tooltip, showing cache event and cache line information.

A lock icon in the column header indicates that the cache way is locked.

A reference map icon in the **Set #** column indicates the results of the reference mapper function. Double-clicking on a cell switches the display to the history view (**History** page) for the selected cell.

History Page

The **History** page (Figure 2-70) displays detailed information for each cache event that occurred in the selected set and way.

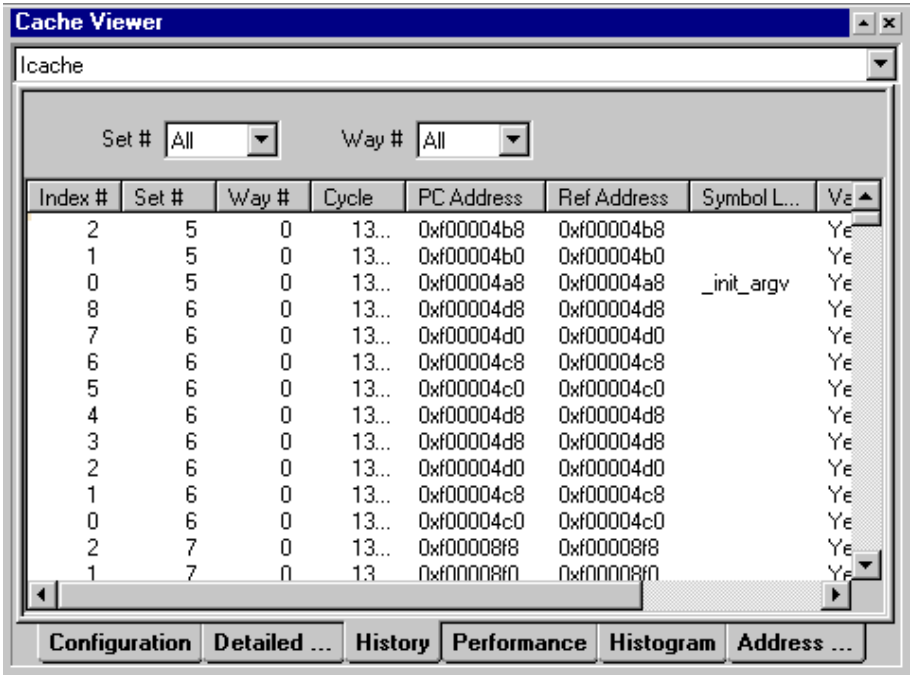

Figure 2-70. Example: History Page

Select the set and way from the pull-down control (top of dialog box) or by double-clicking a cell on the **Detailed View** page.

You can specify the number of stored cache events. Sort the rows by clicking on any column heading. An up arrow in a column heading indicates an ascending sort order; a down arrow indicates a decending sort order.

Table 2-29 describes cache event history information.

Table 2-29. History Information for Cache Events

Item	Description
Index #	Shows the order in which the cache events were received. The index starts at zero and increments each time a cache event is received.
Set #	Displays the set number where the cache event occurred
Way #	Displays the way number where the cache event occurred
Cycle	Displays the cycle count when the cache event occurred
PC Address	Displays the PC address of the cache event
Ref Address	Displays the reference address of the cache event
Symbol Lookup	Displays the symbol name when the reference address resolves to a symbol in memory
Valid	Displays the cache line valid flag (**Yes** or **No**)
Event Type	Displays the cache event type, such as **Hit** or **Miss**
Description	Displays the cache event's description

Performance Page

The **Performance** page (Figure 2-71) shows a list of performance metrics (items and values), which are determined by the target.

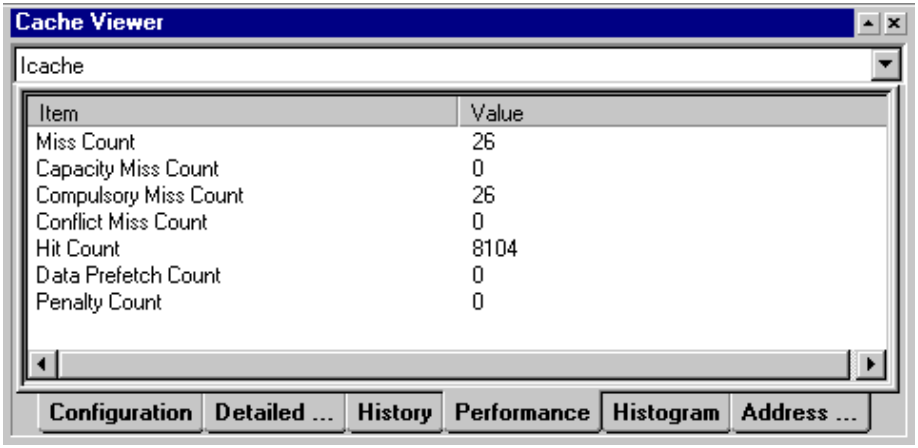

Figure 2-71. Example: Performance Page

The target updates this list. The update rate, however, is not predetermined.

Histogram Page

The **Histogram** page (Figure 2-72) shows a plot of the total number of cache events that occurred in each cache set.

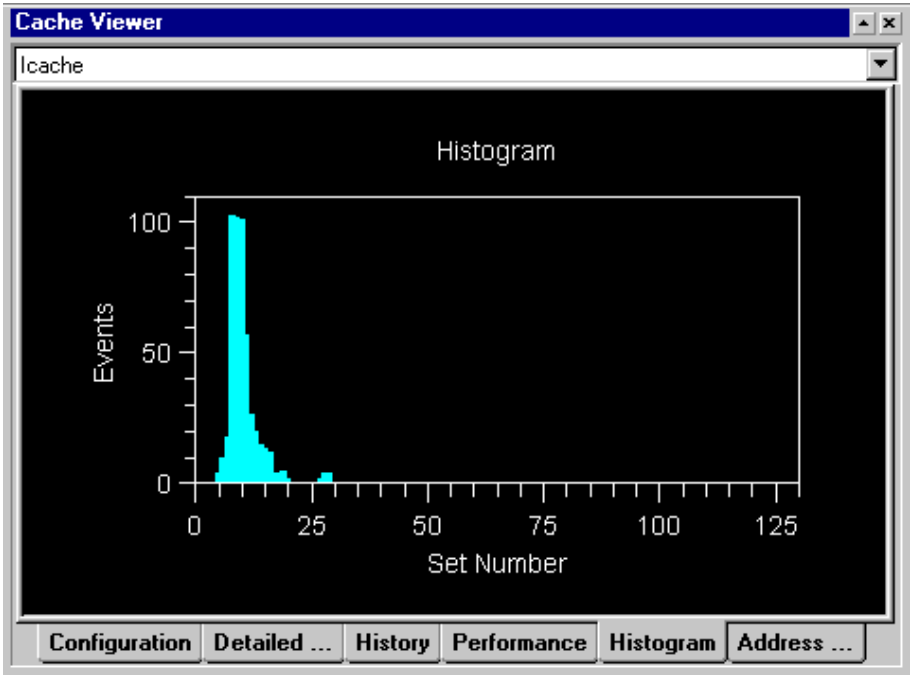

Figure 2-72. Example: Histogram Page

A vertical line displays for each cache set. The line starts at zero and ends at the total number of cache events. Use this plot to identify the most active cache sets.

Address View Page

The **Address View** page (Figure 2-73) displays cache events on an Address versus Cycle plot. Use this view to display the cache events for the specified addresses over time.

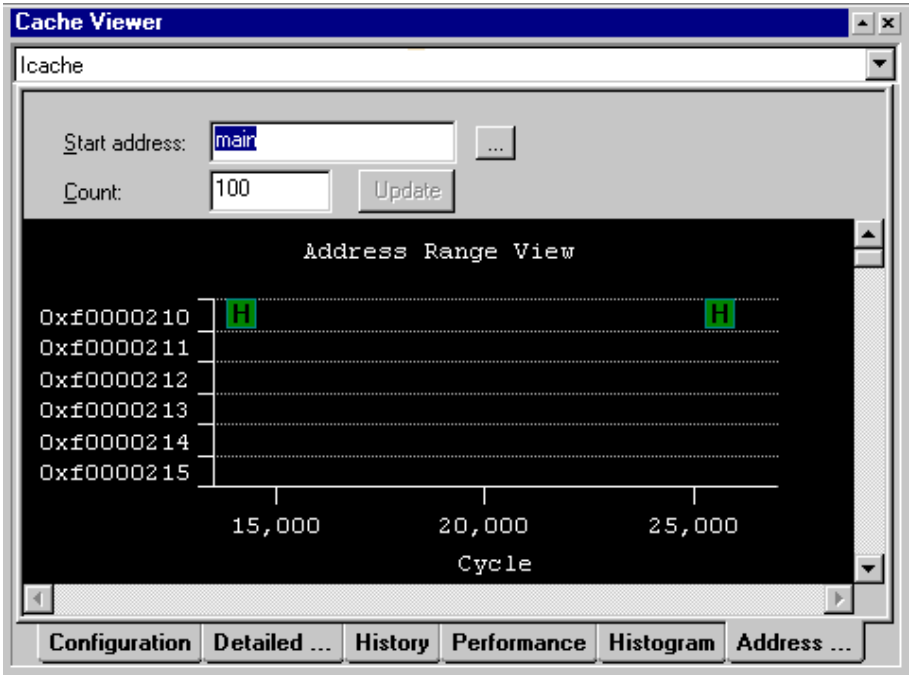

Figure 2-73. Example: Address View Page – Address Range View

Cache events display as icons, identical to the icons used in the detailed view. A start address and count are required. Enter the start address as a hexadecimal value or a symbol. Click the browse (…) button to browse for a symbol.

The count determines the number of addresses displayed. After entering a start address and count, click **Update** to display the cache event data. Use horizontal and vertical scroll bars to scroll the view.

VDK Status Window

The **VDK Status** window (Figure 2-74) is available when an executable file is built with VDK support enabled. Open this window by choosing **View**, **VDK Windows**, and **Status**.

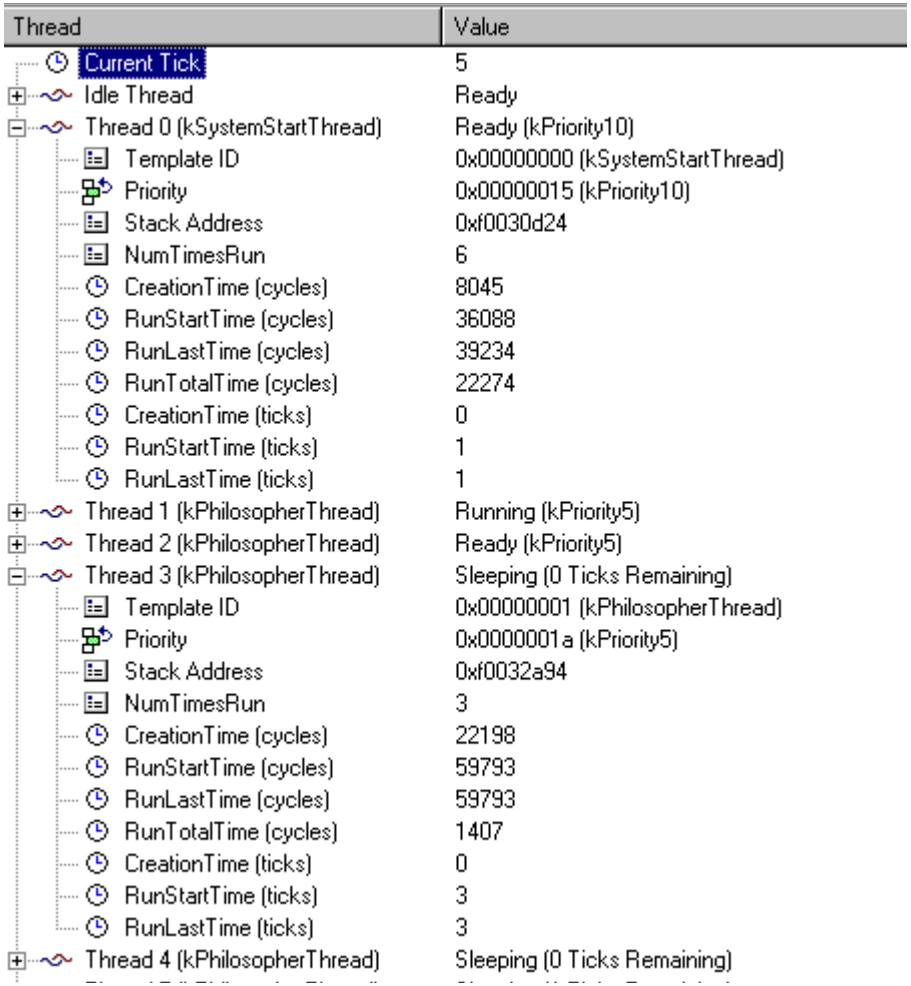

Figure 2-74. Example: VDK Status Window

When the execution of a VDK-enabled program is halted, VisualDSP++ reads data for threads, semaphores, events, event bits, device flags, memory pools, and messages and displays state and status data in this window. When one of the above VDK entities is created, it is added to the display. An entity is removed from the display when it is destroyed.

Initially, information is displayed in a collapsed state, which shows only the name of the entity and, in some cases, its current state. When a thread is in the Ready state, its priority displays.

Clicking the plus sign ( ) next to the name of an entity expands the view.

The possible thread states are as follows.

- Running
- Ready
- SemaphoreBlocked
- EventBlocked
- DeviceFlagBlocked
- MessageBlocked
- SemaphoreBlockedWithTimeout
- EventBlockedWithTimeout
- DeviceFlagBlockedWithTimeout
- MessageBlockedWithTimeout
- Sleeping
- Unknown

See the *VisualDSP++ 4.0 Kernel (VDK) User's Guide* for details.

VDK State History Window

VDK state history is available only for executable files with VDK support. During execution of a VDK-enabled program, if **Full Instrumentation** is specified for the project, thread and event data are collected in a history buffer. When a running program is halted, the history buffer data is plotted in the **VDK State History** window, described in Figure 2-75. Some features become available only when the data cursor is enabled. Open this window by choosing **View**, **VDK Windows**, and **History**.

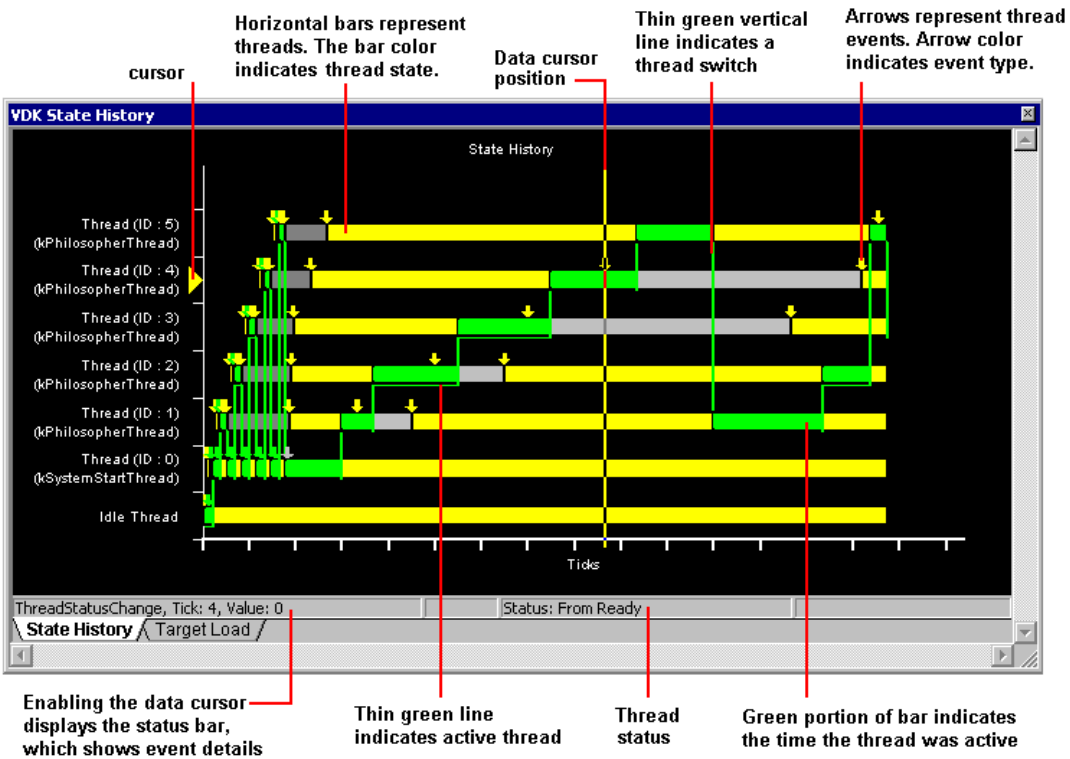
cursor
Horizontal bars represent threads. The bar color indicates thread state.
Data cursor position
Thin green vertical line indicates a thread switch
Arrows represent thread events. Arrow color indicates event type.
VDK State History
State History
Thread (ID : 5) (kPhilosopherThread)
Thread (ID : 4) (kPhilosopherThread)
Thread (ID : 3) (kPhilosopherThread)
Thread (ID : 2) (kPhilosopherThread)
Thread (ID : 1) (kPhilosopherThread)
Thread (ID : 0) (kSystemStartThread)
Idle Thread
Ticks
ThreadStatusChange, Tick: 4, Value: 0
Status: From Ready
State History
Target Load
Enabling the data cursor displays the status bar, which shows event details
Thin green line indicates active thread
Thread status
Green portion of bar indicates the time the thread was active

Figure 2-75. VDK State History Window

Each thread appears as a horizontal bar (thread status bar). The `ThreadID` and the name of the thread type appear to the left of the bar. When a thread is destroyed, the name of the thread type is no longer displayed. Each thread event appears as an arrow above a thread.

Thread Status and Event Colors

Threads and events are coded by color, based on thread status and event type. The colors appear in the horizontal bars (threads) and colored arrows (events) used throughout the plot. Events of the same type are drawn in the same color.

Right-click on the plot and choose **Legend** to display legends that define each color in the **VDK State History** window. To customize colors, right-click on the plot and choose **Properties**.

Trace thread-switched history by following the thin green line, which winds through the display, passing under threads to indicate the running thread at any particular time. When a context switch occurs and changes the running thread, a vertical green line is drawn from the previously running thread to the next running thread.

When you use the data cursor, a yellow triangle to the left of a thread name identifies the currently running thread.

Window Operations

The status bar (at the bottom of the plot) on the **State History** page shows the event's details and thread status when the data cursor is enabled. Event details include the event type, the tick when the event occurred, and an event value. The value for a thread-switched event indicates the thread being switched in or out.

Right-click on the plot and choose **Data Cursor** to activate the data cursor, which is used to display event and thread status details. Based on the event that occurred, the thread status changes. Press the keyboard's right arrow key or left arrow key to move to the next or previous event. When the data cursor hits a thread-switched event, it moves to the thread being switched in. The yellow triangle to the left of the thread name indicates the currently active thread.

You can zoom in on a region to examine that area in more detail. Hold the left mouse button down while dragging the mouse to create a selection box. Then release the mouse button to expand the plot. To restore the plot to its original scale, right-click on the plot and choose **Reset Zoom**.

Right-Click Menu

The **VDK State History** window's right-click menu provides easy access to operations that can be performed from the state history plot.

Target Load Window

Clicking the **Target Load** tab from the **VDK State History** window displays the **Target Load** window. A target load plot (Figure 2-76) shows the percentage of time that the target spent in the Idle thread.

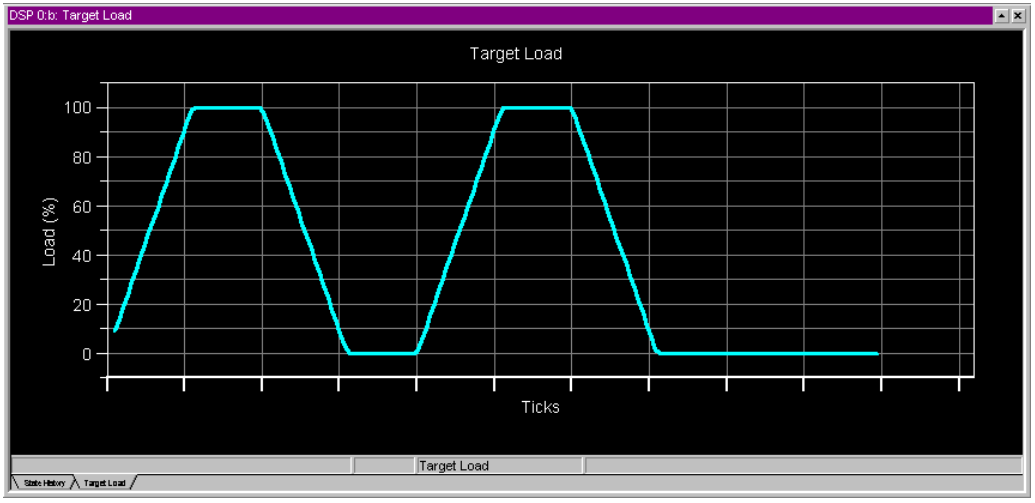

Figure 2-76. Target Load Window Plot

A load of 0% indicates that VDK spent all of its time in the Idle thread. A load of 100% indicates that the target did not spend any time in the Idle thread.

Load data is processed by means of a moving window average.

Plot Windows

Use a plot window to display a *plot*, which is a visualization of values obtained from processor memory. You can display one or multiple plot windows by choosing **View**, **Debug Windows**, **Plot**, and **New**.

In the **Plot Configuration** dialog box, specify the contents of a plot. In the **Plot Settings** dialog box, specify the plot's presentation. You can modify a plot's configuration and immediately view the revised plot.

Figure 2-77 shows an example of a plot window.

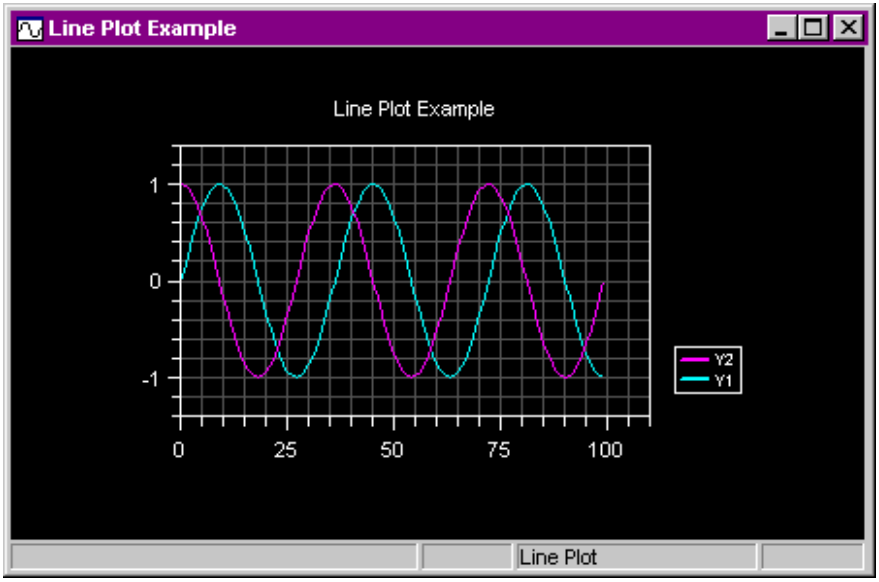

Figure 2-77. Plot Window

From a plot window, you can zoom in on a potion of a plot or view the values of a data point.

You can print a plot, save the plot image to a file, or save the plot's data to a file. For details, refer to VisualDSP++ online Help.

Plot Window Features

Plot windows include a status bar, toolbar, and a right-click menu.

Status Bar

The status bar, located at the bottom of the plot window, displays the plot type and other information, depending on the plot type and other settings.

The following examples show different plot information displayed on the status bar.

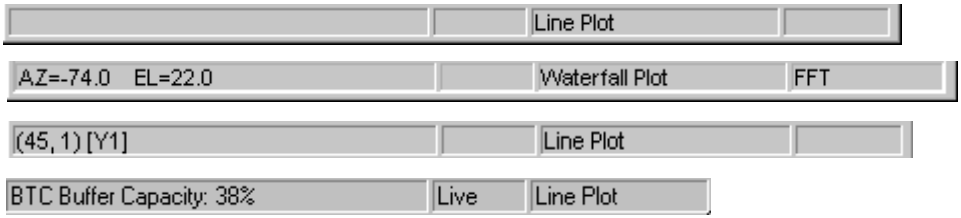

Figure 2-78. Status Bar Information for Plots

In a waterfall plot, the status bar indicates the azimuth and elevation viewing angles. If you zoom in on a region, the status bar indicates that zoom is enabled. When using the data cursor, the status bar shows the selected point's data value.

When a plot window's auto-refresh mode is enabled in BTC mode, the status bar indicates current buffer capacity (for example, 89%) and data logging status.

Buffer capacity, which dynamically changes between 0 and 100%, indicates the portion of the buffer currently in use. The ideal size is a little below 100%. Readings of 100% indicate lost data.

Table 2-30 describes the data logging status indicators in a plot window.

Table 2-30. Data Logging Status Indicators in a Plot Window

Status	Indicates
Record	Real-time data being displayed is also being saved (logged) to a `.BIN` file.
Live	Data is being displayed in real time.
Playback	A previously saved data (log) file is being viewed.

Tool Bar

The plot window's toolbar, shown in Figure 2-79, provides buttons for recording and playing back streaming data and a box for specifying streamed data (`.BIN`) file names.

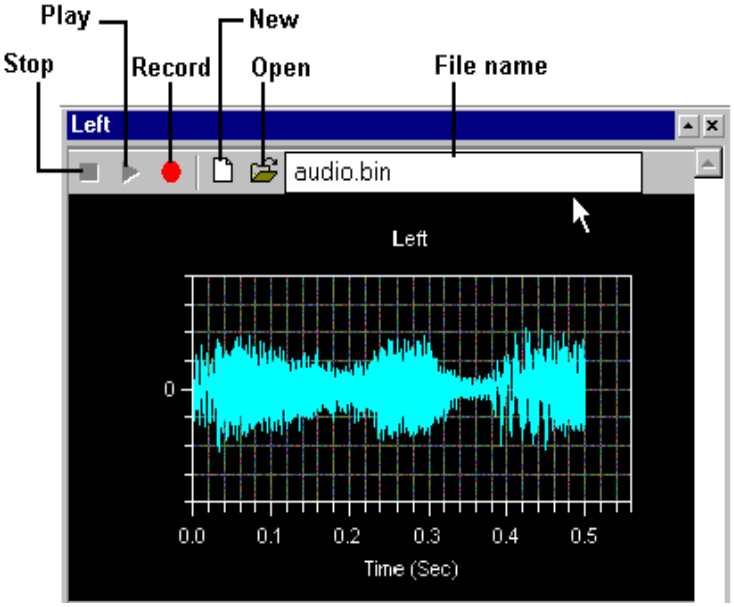

Figure 2-79. Plot Window's Toolbar

Right-Click Menu

The plot window's right-click menu is shown in Figure 2-80.

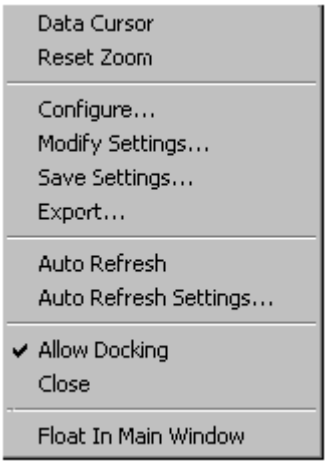

Figure 2-80. Plot Window's Right-Click Menu

This menu provides access to the standard window options (docking, closing, and floating in the main window) and to the plot window features described in Table 2-31.

Table 2-31. Plot Window Operations

Feature	Description
Data Cursor	Displays the data value associated with the position of the plot window's data cursor. View the value on the left side of the plot window's status bar. Press the keyboard's arrow keys to move around the graph.
Reset Zoom	Resets the plot window to its initial full-scale display
Configure	Opens the **Plot Configuration** dialog box, where you add, remove, or modify data sets. You can also change the plot type and rename the plot.
Modify Settings	Opens the **Plot Setting** dialog box, where you customize the plot's appearance. You can specify plot settings (grids, colors, margins, fonts, axes, and so on) and settings for each data set (data processing).
Save Settings	Saves plot configuration settings for future use. The configuration is stored, but not the data. You can retrieve settings (`.VPS` file) and load new plot data.
Export	Exports the plot image to various destinations including the Windows clipboard. Save the plot image as a file (JPG, GIF, TIF, EPS, TXT, or DAT format) or print a hard copy.
Auto Refresh	Enables a plot window to refresh automatically based on settings that you specify. The auto-refresh timer starts. Streaming data is read and displayed. When this option is deselected, the timer is stopped and streaming data is not processed. You can specify auto-refresh options such as BTC, refresh rate, and missing data indication.
Auto Refresh Settings	Enables you to configure options that control auto-refresh settings for plot windows. These settings determine the method in which memory is transferred.

Plot Window Statistics

View various statistics (mean, standard deviation, signal-to-noise ratio (SNR), minimum data value, and maximum data value) while displaying a plot. Note that statistics apply only to the portion of data that is visible. When the plot is zoomed, the statistics are recalculated only for the visible area.

Figure 2-81 shows statistics displayed for a portion of audio data.

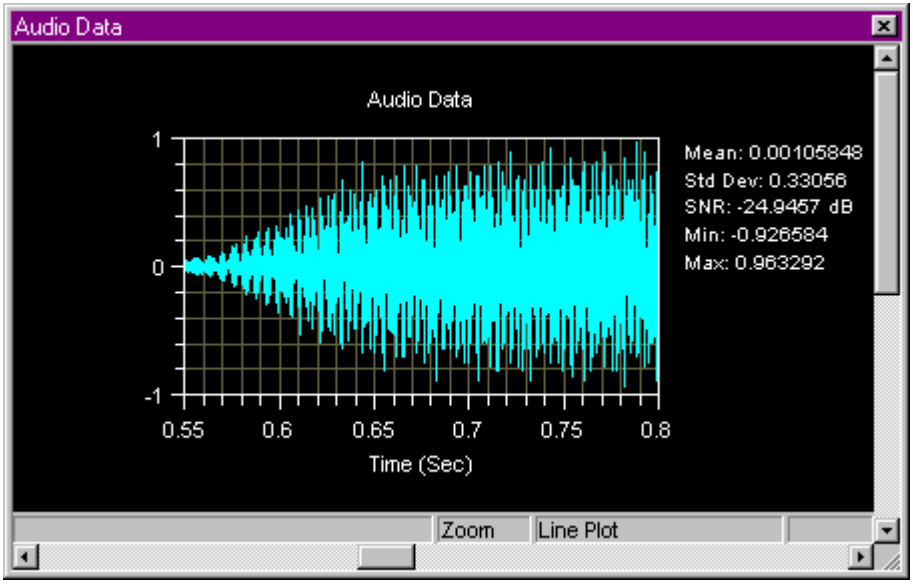

Figure 2-81. Statistics Displayed for a Portion of Audio Data

For details about viewing statistics, refer to VisualDSP++ online Help.

Plot Configuration

A plot configuration comprises two parts: data values and presentation (configuration) settings.

A plot window must contain at least one *data set*, a series of data values in processor memory. Create data sets in the **Plot Configuration** dialog box, shown in Figure 2-82.

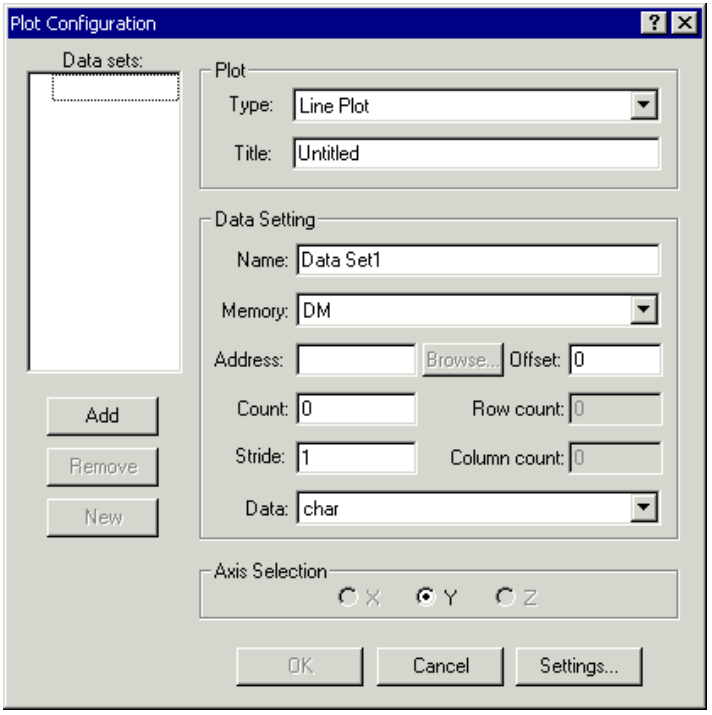

Figure 2-82. Plot Configuration Dialog Box

Specify the type of plot (for example, waterfall), the memory location, the number of values, the axis associated with each data set, and other options that identify the data. Note that three-dimension (3-D) plots require additional specifications for row and column counts.

The **Settings** button enables the configuration of presentation options (such as titles, grids, fonts, colors, and axis presentation) for each data set. You can recall a plot from a saved settings file (`.VPS`). VisualDSP++ uses these settings and reads processor memory to display a plot window.

Plot Window Presentation

Customize the presentation of a plot window to fit your needs. Configure presentation settings from the **Plot Settings** dialog box, which you can invoke by:

- Right-click from within a plot window
- Click the **Settings** button in the **Plot Configuration** dialog box

The **Plot Settings** dialog box provides the tabs shown in Figure 2-83.

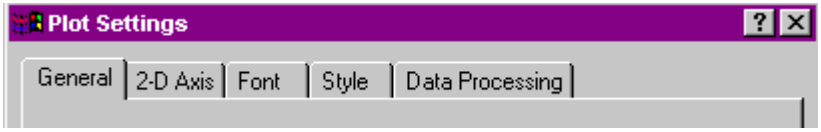

Figure 2-83. Tabs in the Plot Setting Dialog Box

Options on the tab pages enable you to configure the plot window's presentation. On the **Style** page, for example, you can easily specify symbols for a data set as well as line type and width, as shown in Figure 2-84.

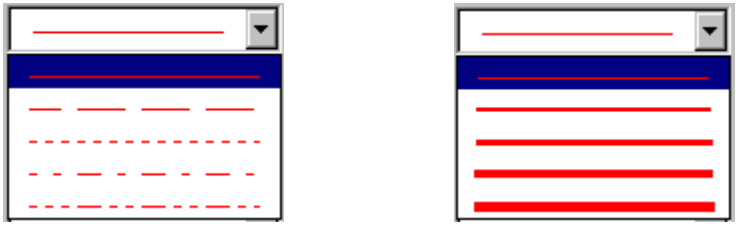

Figure 2-84. Specifying Line Styles

In addition to the many presentation options, you can select a rectangular area, as shown in Figure 2-85, and zoom in on it.

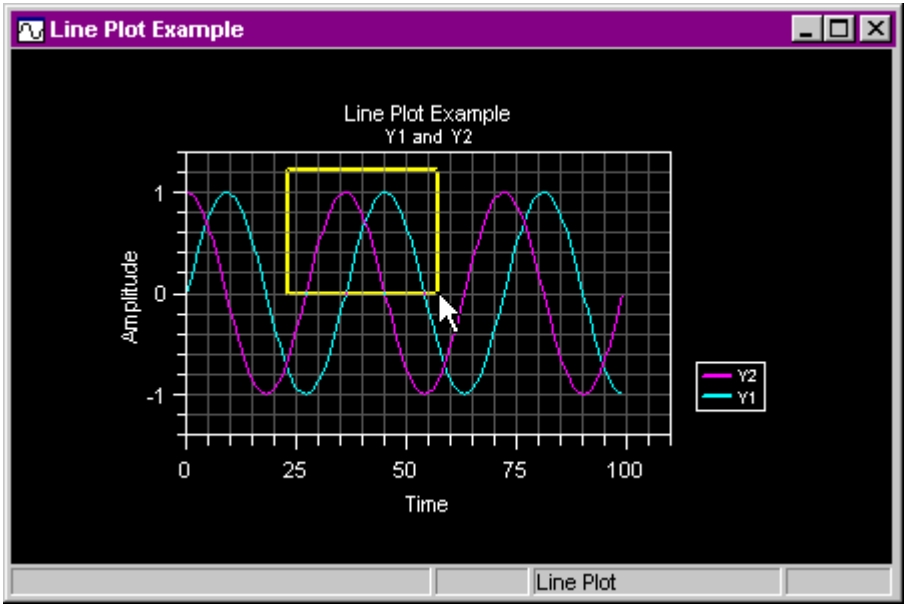

Figure 2-85. Zooming in on a Selected Area

Plot Presentation Options

Depending on the selected plot type, many plot presentation options are available.

In the **Plot Settings** dialog box, these options are grouped by function on tabbed pages, as described in Table 2-32.

Table 2-32. Plot Settings Options by Page

Page	Options That You Can Specify
General	Title and subtitle, grid lines, margins, background colors, and legend
2-D Axis	For X-axis and Y-axis: axis titles, start and increment values, scales
3-D Axis	For X-axis, Y-axis, and Z-axis: axis titles, Z-axis settings, step sizes, scale multipliers, color and mesh
Font	Font name, color, and size
Style	For a data set: line type, width, color; symbol and type
Data Processing	For a data set: data processing algorithm, sample rate, and triggering

You can specify a plot's presentation options before generating the plot (while configuring the plot) or after generating the plot.

Image Viewer

The **Image Viewer** window reads and displays image data from processor memory or a file on your PC. Use this window to configure image attributes and to view images. This display is ideal for testing image-processing algorithms.

You select the image source (from processor memory or a file on your PC) and specify image attributes. If the image is located in processor memory, you must specify the image's address, size, and format.

Tools
Image Viewer
Zoom 2:1
Address:0x10C475 RGB: 0xB7 0x17 0x23 [col:200 row:139]
Zoom status
Status bar indicates the DSP address, RGB values, and pixel's coordinates

Figure 2-86. Image Viewer Window

The Image Viewer supports the following pixel formats: Grayscale 8, Grayscale 12, Grayscale 16, RGB555, RGB565, RGB24, and RGB32.

To open the **Image Viewer** window, choose **View**, **Debug Windows**, and **Image Viewer**.

Refer to online Help for information about format types, packed data, and detailed how-to information.

Automation Interface

The Image Viewer has an automation interface that permits COM-aware languages to access Image Viewer functions, such as loading, retrieving, displaying, and saving data.

Toolbar

The top of the window provides these tools:

Table 2-33. Image Viewer Window Toolbar Buttons

Button	Purpose
	Configure. Opens the Image Configuration dialog box
	Refresh. Reads processor memory and updates the image display
	Zoom In. Zooms in by a factor of two
	Zoom Out. Zooms out by a factor of two
	Zoom Cancel
	Save. Opens a Save dialog box, from which to save the image.
	Print
	Copies the image to the Windows clipboard

Status Bar

As you move the mouse over the image, the status bar indicates:

- Zoom status
- Processor address where the selected pixel is located
- Pixel values for color images, intensity values for gray-scale images
- Pixel coordinates (column and row)

Pixel color depth is 24 bits for color images and 8 bits for gray-scale images.

Right-Click Menu

The **Image Viewer** window's right-click menu provides these commands:

Table 2-34. Right-Click Menu Commands

Command	Purpose
Configure	Opens the **Image Configuration** dialog box, from which to specify image attributes
Refresh	Reads the image data from processor memory
Color - Gamma Adjust	Opens the **Image Effects** dialog box, from which to adjust gamma and view the resulting image
Rotate	Provides four selections: **0**, **90**, **180**, or **270**
Flip	Provides four selections: **None**, **Horizontal**, **Vertical**, or **Both**.
Auto-Refresh	A black check mark indicates that auto-refresh is enabled (based user-specified settings)
Auto-Refresh Settings	Opens the **Auto-Refresh Settings** dialog box, from which to configure auto-refresh settings

3 DEBUGGING

This chapter describes VisualDSP++ debugging tools used during single-processor and multiprocessor debug sessions. The topics are organized as follows.

- "Debug Sessions" on page 3-2
- "Code Analysis Tools" on page 3-7
- "Program Execution Operations" on page 3-10
- "Simulation Tools" on page 3-16
- "Plots" on page 3-18
- "Flash Programmer" on page 3-27

Debug Sessions

You run the projects that you develop as *debug sessions* (sessions).

A session is defined by the elements described in Table 3-1.

Table 3-1. Specifying a Debug Session

Element	Description
Debug target	The debug target is the software module that controls a type of debug target (a simulator or emulator). The *simulator* is software that mimics the behavior of a processor chip. Simulators are used to test and debug processor code before a processor chip is manufactured. An *emulator* is software that "talks" to a hardware board that contains one or more actual processors.
Platform	For a given debug target, several platforms may exist. For a simulator, the platform defaults to the identically named simulator. When the debug target is an EZ-ICE® board, the platform is the board in the system on which to focus. When the debug target is a JTAG emulator, the platforms are the individual JTAG chains.
Processor	Multiple processors can exist for a given debug target and platform. When you create an executable file, the processor is specified by the Linker Description File (`.LDF`) and other source files.

When setting up a session, set the focus on a series of more specific elements.

The target platform and processor settings specify the debug session. A default session name is automatically generated. You can further identify the session by modifying the default name, choosing a more meaningful name.

A well-chosen name can prevent confusion later.

Debug Session Management

You can run several debug sessions at once and can dynamically switch between sessions.

The typical reasons for running multiple debug sessions are:

- To write different versions of your program to compare their operating efficiencies
- To debug completely different programs without having to run multiple instances of VisualDSP++

Simulation vs. Emulation

While connected to a simulator session, you may open as many sessions as your system's memory can handle.

While connected to actual hardware through an emulator, only **one** debug session may be connected to one emulator at any time. If multiple emulators are installed and are connected to multiple target boards, one debug session may be connected to each individual emulator.

In a JTAG emulator session, only one debug session may be connected to each physical target/emulator combination. Otherwise, contention issues may arise. Upon switching to a different session, VisualDSP++ detaches from the old session before attaching to the new session.

Breakpoints

In a simulator session, a breakpoint can be set at any address in your executable program's memory. Program execution halts at the address where the breakpoint is located.

Hardware breakpoints can be used in emulator debug sessions only.

Watchpoints

Watchpoints are like breakpoints that trap on a specified condition.You can set watchpoints on registers, stacks, and memory ranges. Reaching the condition halts program execution and updates all windows.

Watchpoints are available only during simulation.

Multiprocessor (MP) System Debugging

Often, performance-based products require two or more processors. A system built with multiple processors is called a multiprocessor system (MP system). A system built with a single processor is called a single-processor system.

The SHARC and TigerSHARC simulators do not support multiprocessor (MP) debugging; multiprocessing for these processors is available in emulator sessions only. Multiprocessing support for Blackfin ADSP-BF561 and ADSP-BF566 processors is available in simulation.

Setting Up a Multiprocessor Debug Session

The first step in setting up a multiprocessor debug session is to develop a multiprocessor project by using the multiprocessing capabilities of the linker and an `.LDF` file to describe the multiprocessor system. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual*, especially sections about the `SHARED_MEMORY{}` and `MPMEMORY{}` commands.

The second step is to use the VisualDSP++ Configurator utility to describe the hardware to the VisualDSP++ software if you are running a JTAG emulator session. VisualDSP++ uses this description when you set up your debug session. Refer to your processor's hardware documentation for information required when you use the VisualDSP++ Configurator.

When running a multiprocessor simulator debug session, select the desired configuration from the **Platforms** list in the **New Session** dialog box. After specifying your hardware system, build your project.

The first time that you launch VisualDSP++ for a new project, the **New Session** dialog box opens to enable you to configure the MP session. The next time VisualDSP++ is launched, the debug session is configured automatically.

Debugging a Multiprocessor System

Debugging a multiprocessor system requires that you synchronously run, step, halt, and observe program execution operations in all the processors at once.

The following capabilities help to speed a multiprocessor debug session.

- Multiprocessor debug commands operate similar to commands used to debug a single processor. The only difference is that MP commands work synchronously on *all active processors* in the currently selected MP group
- **Multiprocessor** window (refer to "Multiprocessor Window" on page 2-90)

  The **Status** page displays the status of each processor and lets you switch processor focus.

  The **Group** page enables you to group processors into multiple, logical units to which all MP commands are applied.
- Window pinning. Note that you can use pinning and the processor status items in the **Multiprocessor** window with single-processor debug commands to debug individual processors in an MP session.
- Window color specification

Focus and Pinning

In a multiprocessor debug session, you often have to examine the behavior of a single processor to better understand its interaction with the other processors on the target.

When you debug a single processor in an MP session, the processor being debugged has the *focus*.

By *pinning* a window to a processor, you dedicate that window (such as a memory window) to a particular processor in a multiprocessor group. Pinning statically associates a window to a specific processor.

Before debugging, open and pin the register windows and memory windows that you plan to use. If these windows are not pinned, they display information for any processor that has focus.

When a window is pinned to a processor, a pin icon appears in the window's upper-left corner.

For example:

Window Title Bar Information

Figure 3-1 shows a pinned window in a multiprocessor debug session.

The title bar of a pinned window shows:

- Processor name
- Pushpin icon ( ) to indicate a pinned window
- Window title
- Number format, such as hexadecimal (for windows that support multiple formats)

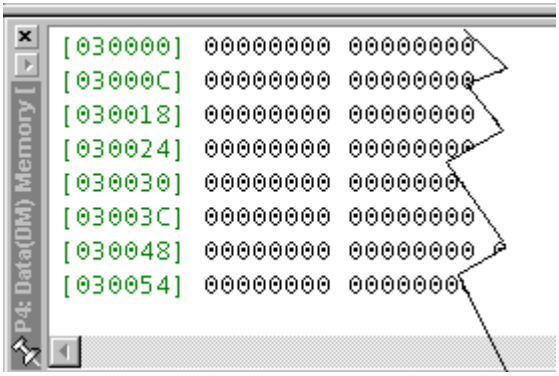

Figure 3-1. Pinned Window in a Multiprocessor Debug Session

Additional Focus Indication

If configured, VisualDSP++ shades unfocused windows with a specified color. You can specify the background color of focused and unfocused windows.

Code Analysis Tools

You use code analysis tools to examine your code's behavior and locate areas that may be optimized for better performance.

VisualDSP++ provides these code analysis tools:

- Statistical profiles and linear profiles
- Traces
- Processor memory plots

Statistical Profiles and Linear Profiles

VisualDSP++ provides two profiling methods that measure program performance by sampling the target's Program Counter (PC) register to collect data. During program development use linear profiling with simulator targets, and use statistical profiling with emulator targets.

The **Linear Profiling Results** window and **Statistical Profiling Results** window display the data collected by these two profiling methods and indicate where the application is spending its time. Refer to "Statistical/Linear Profiling Results Window" on page 2-63for details.

The window's title (**Linear Profiling Results** or **Statistical Profiling Results**) depends on whether this tool is used during simulation or emulation.

Simulation: Linear Profiling

Linear profiling with the simulator is not statistical because the simulator samples *every* PC executed. This feature provides an accurate and complete picture of program execution.

Linear profiling is much slower than statistical profiling. Simulator targets support linear profiling, but do not support statistical profiling.

Emulation: Statistical Profiling

A statistical profile measures the performance of a user program by sampling the target's PC register at random intervals while the target is running the program. Most of the execution time in the program is in the areas where most of the PC registers are concentrated.

Statistical profiling provides a more generalized form of profiling that is well suited to JTAG emulator debug targets. Emulator targets do not support linear profiling.

JTAG sampling is completely non-intrusive, so the process does not incur additional run-time overhead.

Traces

A trace captures a history of processor activity during program execution. Run a trace (also called an execution trace or a program trace) to analyze the run-time behavior of your application program, enable I/O capabilities, and simulate source to target data streaming.

TigerSHARC processors do not support traces.

VisualDSP++ provides a Trace window to help you analyze program run-time behavior. Refer to "Trace Windows" on page 2-60 and to online Help for details.

A trace includes the following information.

- Buffer depth (instruction lines)
- Cycle count
- Instructions executed such as memory fetches, program memory writes, and data/memory transfers

Viewing the disassembled instructions that were performed can also help in analyzing code behavior.

Program Execution Operations

When VisualDSP++ starts up, it attaches to the previous session by default. You can override this behavior, and instead, force VisualDSP++ to start a new session.

When loading and running your program, use VisualDSP++ features to step, break, and halt the program.

Selecting a New Debug Session at Startup

If you had a problem, such as a corrupted workspace, in your last debug session, use the following procedure to force a fresh session at startup.

VisualDSP++ must be closed before performing the following procedure.

1. Hold down the keyboard's **Ctrl** key.

   Do **not** release the **Ctrl** key until the **Session List** dialog box appears, as described in the next step.

2. Invoke VisualDSP++ as you normally do.

   Typical methods include using the Windows **Start** button sequences, clicking desktop icons, or using Windows Explorer.

   The **Session List** dialog box appears.

3. Specify and activate a debug session.

   When launching VisualDSP++ in stand-alone mode, ensure that the session is configured correctly *before* loading the program.

Loading the Executable Program

Once you have specified the debug session, begin the session by loading the executable program.

After a successful build of the target executable program, VisualDSP++, if configured, loads the program automatically to the current session when the session processor type matches the project's processor. If the current session processor does not match the project's processor type, you are prompted to choose another session.

If automatic load is not configured, VisualDSP++ does not try to load the executable program automatically after a successful build.

The target must be an executable (`.DXE`) file.

This debugging feature saves time, as you do not have to load the executable target manually, and you can start to debug immediately after successfully building the project.

Program Execution Commands

Run program execution commands from the **Debug** menu or from the toolbar.

Executable files run until an event such as a breakpoint, watchpoint, or user-issued **Halt** command stops execution. When program execution halts, all windows are updated to current addresses and values.

Multiprocessor (MP) commands operate like single-processor commands with one exception—they perform an action on *all* processors in the MP group. **MP Run**, **MP Halt**, and **MP Step** are synchronous operations, which means that all processors in the currently selected MP group execute on exactly the same clock cycle. Some commands have keyboard shortcuts; refer to "Keyboard Shortcuts" on page A-32.

Use the commands described in Table 3-2 to control program execution.

Table 3-2. Commands Used to Control Program Execution

Command	Description
Run	Runs an executable program. The program runs until an event stops it, such as a breakpoint or user intervention. When program execution halts, all windows update to current addresses and values.
Halt	Stops program execution. All windows are updated after the processor halts. Register values that changed are highlighted, and the status bar displays the address where the program halted.
Run to Cursor	Runs the program to the line where you left your cursor. You can place the cursor in editor windows and **Disassembly** windows.
Step Over	(C/C++ code only in an editor window) Single-steps forward through program instructions. If the source line calls a function, the function executes completely, without stepping through the function instructions.
Step Into	(editor window or **Disassembly** window) Single-steps through the program one C/C++ or assembly instruction at a time. Encountered functions are entered.
Step Out Of	(C/C++ code only in an editor window) Performs multiple steps until the current function returns to its caller, and stops at the instruction immediately following the call to the function.

Restarting the Program

You can set the Program Counter (PC) to the first address of the interrupt vector table.

Performing a Restart During Simulation

In the simulator, restart works like a reset; however, the target's memory does not change. All registers are reset to their initial values.

Memory is not reset. Thus, C and assembly global variables are *not* reset to their original values. Your program may behave differently after a restart. To re-initialize these values, reload your `.DXE` file.

Performing a Restart During Emulation

In the emulator, a restart works exactly like a reset. Only registers with default reset values are affected. All other registers remain unchanged.

Breakpoints

An enabled breakpoint halts program execution at a specific instruction or address. You can enable and disable breakpoints as well as add and delete breakpoints.

A disabled breakpoint is set up, but not turned on. A disabled breakpoint does not stop program execution. It is dormant and may be used later.

A *break* occurs when the conditions that you specify are met.

You can quickly place an unconditional breakpoint at an address in a **Disassembly** window or editor window by:

- Selecting an address and clicking **Toggle Breakpoint** button
- Double-clicking a line in a **Disassembly** window or editor window

Symbols in the left margin of a **Disassembly** window or editor window indicate breakpoint status, as shown in Table 3-3.

Table 3-3. Breakpoint Status Symbols

Symbol	Indicates
	An enabled (set) breakpoint
	A disabled breakpoint (recognized, but cleared)

Unconditional and Conditional Breakpoints

A breakpoint configured to occur when the Program Counter reaches a specific address is an *unconditional breakpoint*. The breakpoint occurs when it is reached.

A breakpoint configured to occur when various conditions (criteria) are met is called a *conditional breakpoint*. The conditions may include:

- A user-defined *expression* that must evaluate to TRUE
- A *skip* (count) that specifies the number of times to skip over the breakpoint before finally halting

If both an expression and skip are set, execution stops when the breakpoint is reached "*n*" times when the expression is TRUE, where *n* represents the skip count. When the expression is empty, execution stops when the breakpoint is reached "*n*" times.

Watchpoints

Similar to breakpoints, watchpoints stop program execution when user-specified conditions are satisfied. Watchpoints, however, are used to set a *condition,* such as a memory read or stack pop, for halting events.

Use watchpoints only during simulation.

Watchpoints, unlike breakpoints, are not attached to a specific address. A watchpoint halts anywhere in your program once the watchpoint conditions are satisfied.

Hardware Breakpoints

Similar to simulator watchpoints, hardware breakpoints enable you to set breaks on instructions or data transfers within a user-defined memory range.

Choosing **Hardware Breakpoints** from the **Settings** menu opens the **Hardware Breakpoints** dialog box, where you can configure the following attributes for three breakpoints (**BP0**, **BP1**, or **BP2**).

- **Mode** – type of breakpoint (Disabled, Single Address, In Range, Out of Range)
- **Start Address** and **End Address** – address range in which to break. Specified addresses are inclusive.
- **Access** – type of memory access to which the breakpoint responds (Read Only, Write Only, or In Range)
- **Counter** – number of breakpoint events ignored before the processor stops. Each time this breakpoint condition occurs, its count is decremented. When the count reaches zero (0), the processor processes the breakpoint and halts.
- **Restore Counter on Break** – resets the counter to the counter value each time the breakpoint is processed
- **AND Breakpoints**, **OR Breakpoints** – AND decrements the counter each time all breakpoints find a match in the same cycle. OR decrements the counter when one breakpoint finds a match.

Latency

Hardware breakpoints do not assert until one (1) or two (2) instruction cycles after the actual break condition occurs. Note that the program counter is not placed on the instruction that caused the break.

Restrictions

When using hardware breakpoints, do not place breaks at any address where a `JUMP`, `CALL`, or `IDLE` instruction would be illegal.

Do not place breaks in the last few instructions of a `DO LOOP` or in the delay slots of a delayed branch. For more information on these illegal locations, refer to your processor's hardware documentation.

Simulation Tools

Before you have the processor, you can use interrupts and data streams within VisualDSP++ to simulate the processor's behavior.

Interrupts

Use interrupts to simulate external interrupts in your program. When you use interrupts with watchpoints and streams, your program simulates real-world operation of your processor system.

Input/Output Simulation (Data Streams)

In many products, processors exist as part of a larger system where they can act as a host or a slave. They can drive other devices or take part in processing a subset of data. Because of their extensive I/O capabilities, Analog Devices processors excel in these roles.

Use data streams to transmit data between:

- A device and a file
- A device and a device
- A device in one processor and a device in another processor in a multi-processor system

Using background telemetry channel (BTC) technology, VisualDSP++ permits the streaming of data from a target processor without halting the processor.

This capability applies to emulation targets only.

The plot window receives and displays a stream of data from processor memory. If the target supports background telemetry, the plot window reads memory and updates the display without halting the target. Otherwise, the plot window halts the processor, reads memory, updates the plot, and resumes the processor.

The plot window allows data to be streamed to (or from) a binary data file. The data file can be converted into ASCII format for input to other applications such as MATLAB and Excel.

The processor application may collect and transfer data in four different ways:

- Sampling a test point over time
- Transferring a data array over BTC at a specified point in the application (SHARC and Blackfin emulation targets only)
- Using `GetMem()` directly
- Periodically halting the target to read memory

For information about using BTC, see the *VisualDSP++ 4.0 Getting Started Guide*.

Plots

Use the VisualDSP++ data plotting capability to display data in processor memory as a *plot* (graph) in a plot window (Figure 3-2). A data plot can assist you by allowing you to visualize data.

Refer to "Plot Windows" on page 2-115 for plot window configuration information. For complete details on configuring plots, refer to VisualDSP++ online Help.

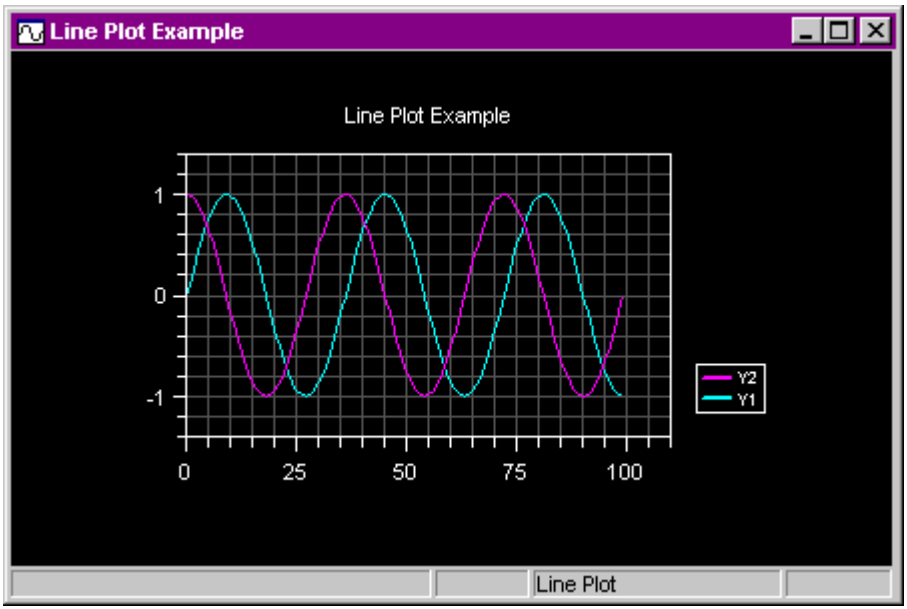

Figure 3-2. Plot Window Displaying Processor Memory

When plotting processor memory, you can:

- Choose from multiple plot types and specify the plot's data and presentation
- Apply a data processing algorithm to the processor's memory data
- Modify a plot's configuration and immediately view the results
- Zoom in on a plot area or view data point values, in a plot window
- Print a plot, save the plot image to a file, or save the plot's data to a file

Plot Types

Each plot must be specified to be one of the plot types in Table 3-4.

Table 3-4. Available Plot Types

Plot Type	Description	Requires
Line (on page 3-20)	Displays points connected by a line	Y value for each data point
X-Y (on page 3-21)	Similar to a line plot, but also uses X-axis data	X value and Y value for each data point
Constellation (on page 3-22)	Displays a symbol at each data point	X value and Y value for each data point
Eye diagram (on page 3-23)	Typically used to show the stability of a time-based signal	Y value for each data point
Waterfall (on page 3-24)	3-D plot typically used to show the change in frequency content of signal over time	Z value for each data point
Spectrogram (on page 3-26)	2-D plot displays amplitude data as a color intensity	Z value for each data point

The X, Y, and Z values are read from processor memory.

Line Plots

A line plot (Figure 3-3) displays a range of processor memory values connected by a line. The values read from processor memory are assigned to the Y-axis. The corresponding X-axis values are automatically generated.

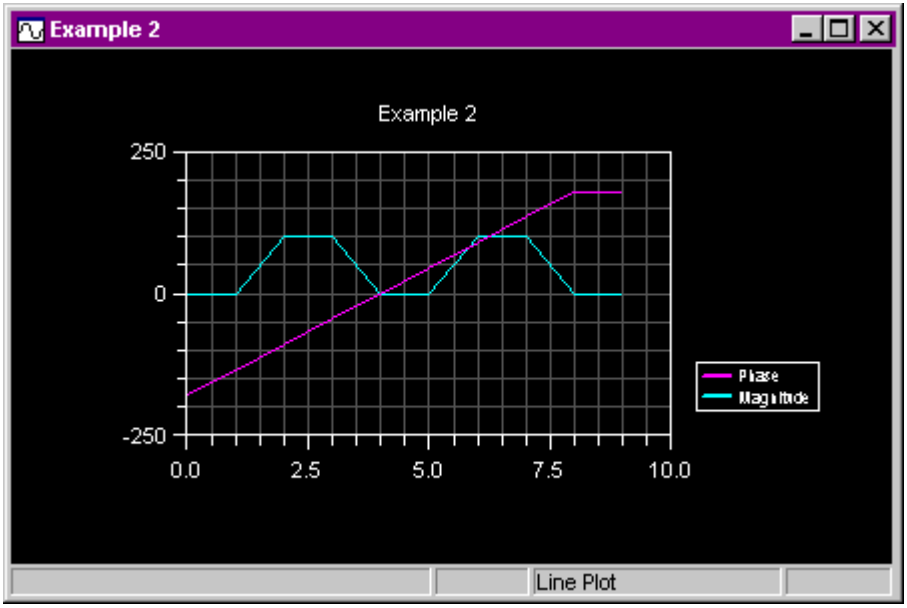

Figure 3-3. Line Plot

Multiple data sets can be plotted on a single graph.

X-Y Plots

An X-Y plot (Figure 3-4) requires an X value and a Y value for each data point. Unlike a line plot, an X-Y plot requires X-axis data.

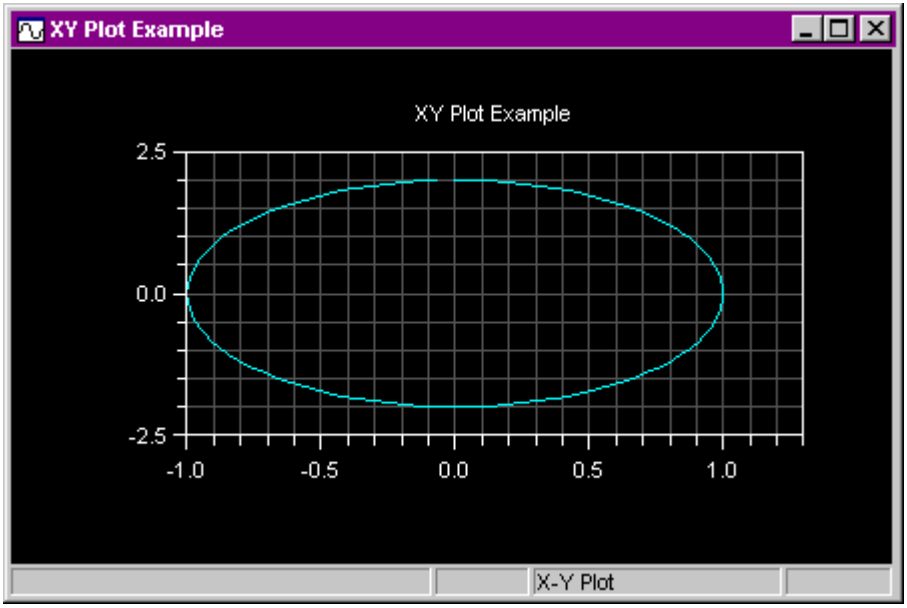

Figure 3-4. X-Y Plot

The X and Y data are specified separately in a user-defined memory location. The number of X and Y points must be equal.

Constellation Plots

A constellation plot (Figure 3-5) displays a symbol at each (X,Y) data point.

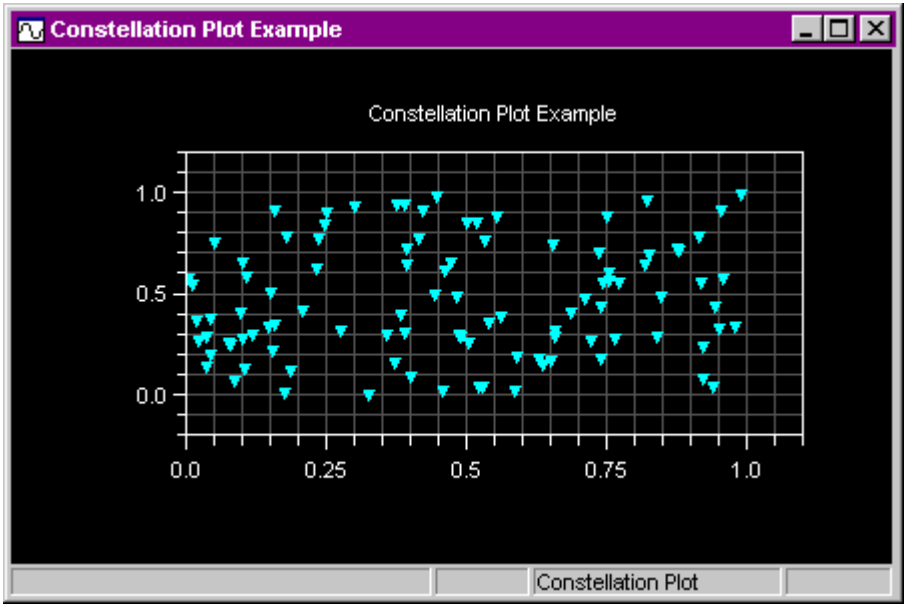

Figure 3-5. Constellation Plot

The X and Y data are specified separately in a user-defined processor memory location. The number of X and Y points must be equal.

Eye Diagrams

An eye diagram plot (Figure 3-6) is typically used to show the stability of a time-based signal. The more defined the eye shape, the more stable the signal.

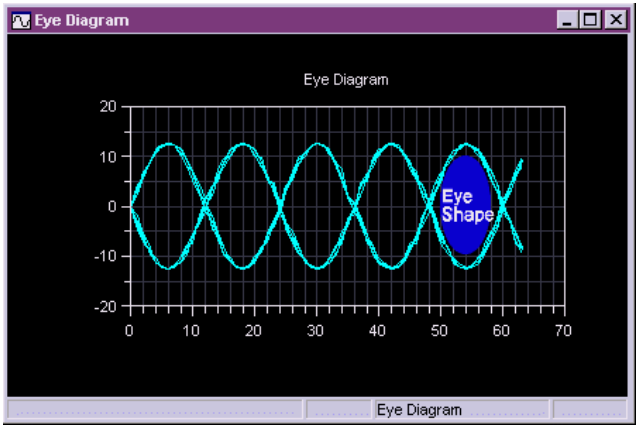

Figure 3-6. Eye Diagram Plot

This plot works like a storage oscilloscope by displaying an overlapped history of a time signal. The eye diagram plot processes the input data and optionally looks for a threshold crossing point (default is 0.0). A trace is plotted when the threshold crossing is reached. Plotting continues for the remainder of the trace data.

When a breakpoint occurs (or a step is performed), the plot data is updated and a new trace is plotted. The eye diagram uses a data shifting technique that stores the desired number of traces in a plot buffer (default is ten traces). When the number of traces is exceeded, the first trace shifts out of the buffer and the new trace shifts into the last buffer location. This technique is referred to as *first in, first out* (FIFO).

You can modify options for threshold value, rising trigger, falling trigger, and the number of overlapping traces.

Waterfall Plots

A waterfall plot (Figure 3-7) is typically used to show the change in frequency content of signal over time.

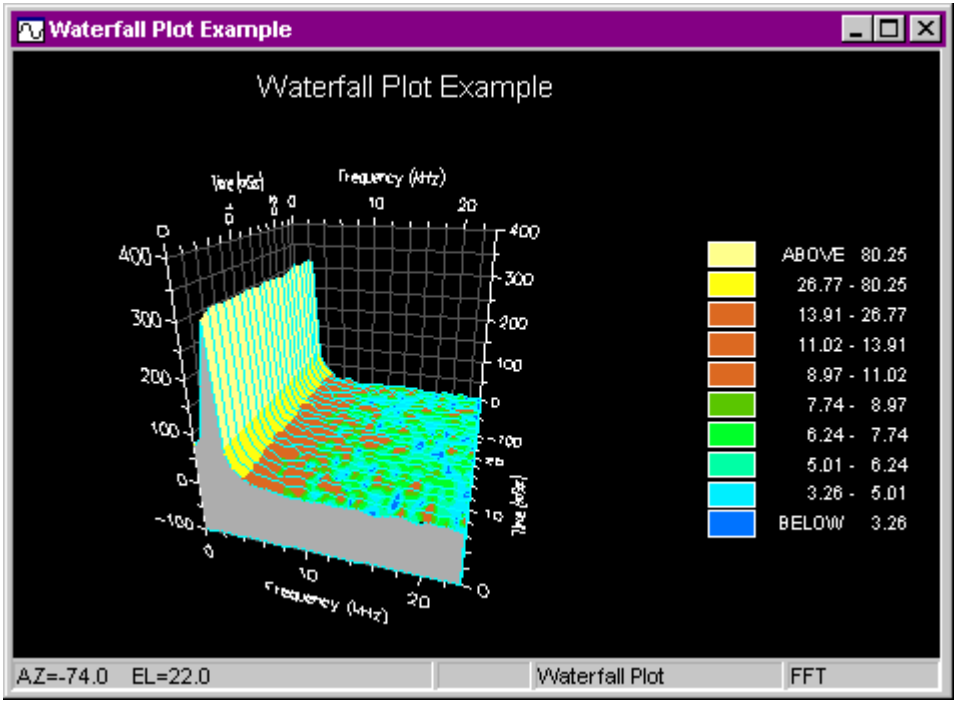

Figure 3-7. Waterfall Plot

The plot comprises multiple line plot traces in a three-dimensional (3-D) view. Each line plot trace represents a slice of the waterfall plot.

The easiest way to create a waterfall plot is to define a 2-D array in C code (a grid). The first array dimension is the number of rows in the grid, and the second dimension is the number of columns in the grid. The number of columns is equal to the number of data points in each line trace.

A time-based signal is sampled at a predefined sampling rate and stored as a slice in the grid (row 0, columns 0–*N*).

Figure 3-8 shows a grid of sampled data.

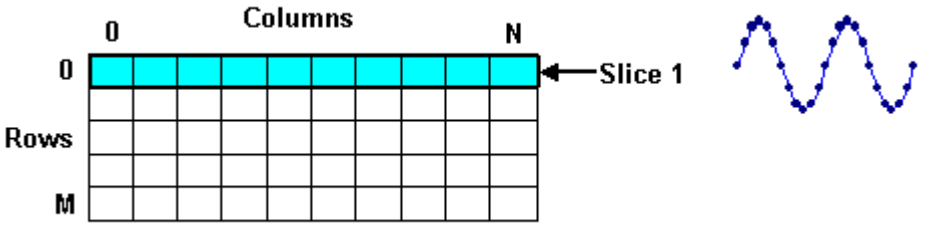

Figure 3-8. Grid of Sampled Data

The next time, a signal is sampled and stored (in row 1, columns 0–*N*). This process continues until all the rows are filled.

By default, an FFT performed on each slice results in a frequency output display. Use a color map on the **3-D Axis** page of **Color Settings** dialog box to enhance the display. Each color corresponds to a range of amplitude values.

The plot output displays a legend showing each color and associated range of values.

You can rotate the waterfall plot to any desired azimuth and elevation by using the keyboard's arrow keys.

Spectrogram Plots

A spectrogram plot (Figure 3-9) displays the same data as a waterfall plot, except in a two-dimensional (2-D) format.

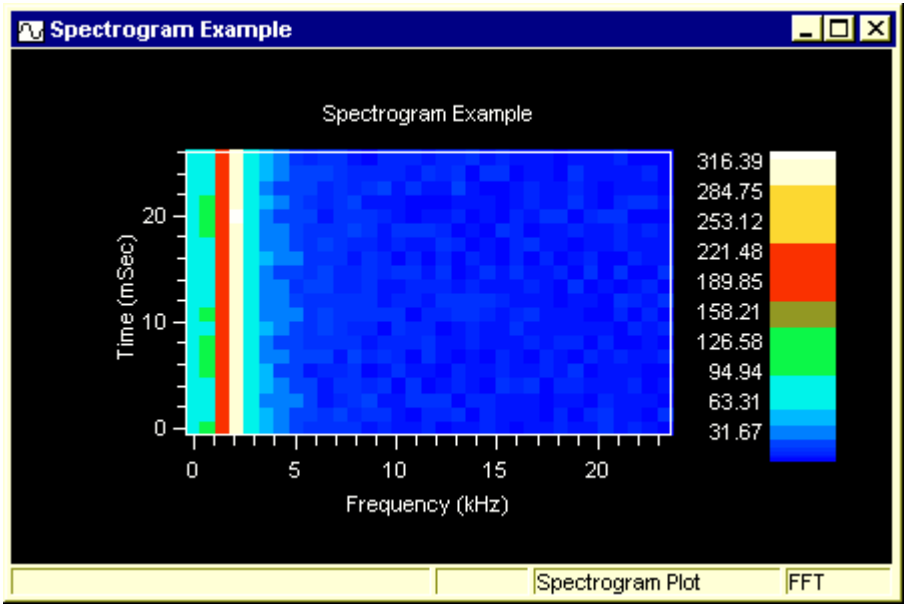

Figure 3-9. Spectrogram Plot

Each (X,Y) location displays as a color representing the amplitude of the data. By default, an FFT performed on each slice results in a frequency output display. A legend displays the colors and associated range of values.

Flash Programmer

The VisualDSP++ Flash Programmer provides a convenient, generic interface to numerous processors and flash memory devices. This utility simplifies the process of changing data values on a flash device and modifying its memory. You no longer have to remove the flash memory from the board, use a separate Flash Programmer, and then replace the flash.

For complete details on using the Flash Programmer utility, refer to VisualDSP++ online Help.

Flash Devices

Flash memory parts are non-volatile memories that can be read, programmed, and erased. In most applications, flash devices store:

- Boot code that the processor loads at startup
- Data that must persist over time and through the loss of power

Flash device programming is typically performed with a device programmer at the factory or by the application developer. When a flash device is wired appropriately to the processor, the processor can program the flash device.

Flash Programmer Functions

Use the Flash Programmer to:

- Load a flash algorithm (driver) program onto the processor at any time
- Obtain the flash manufacturer and device codes
- Reset the flash
- Program the flash from an Intel hex data file
- Fill portions of flash memory with a value and quickly “punch-in” data

- Erase the entire flash
- Erase a single sector
- Send custom commands to the driver for batch processes or user-defined behavior

The utility stores the most recently used information in the registry for retrieval when the utility is next started up. A status indicator shows the utility's current state.

Flash Driver

To use the Flash Programmer utility, you must first load a flash driver onto the processor. The driver is a processor application that interfaces with the Flash Programmer and performs all the interaction with the flash device. Analog Devices supplies sample drivers for use on certain EZ-KIT Lite evaluation systems.

Flash Programmer Window

Figure 3-10 shows the **Flash Programmer** window.

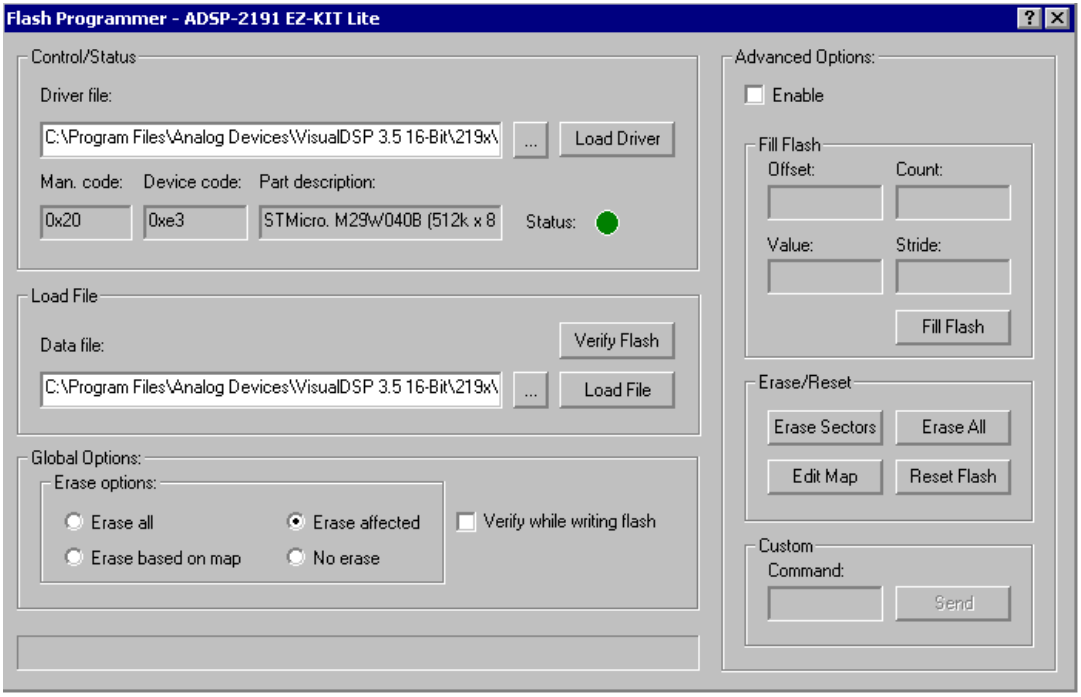

Figure 3-10. Flash Programmer Window

Table 3-5 describes the fields and buttons in the **Flash Programmer** window.

Table 3-5. Flash Programmer Window Controls

Control	Description
Control/Status	
Driver file	Specifies the path and name of the driver file. Type the path and file name or browse to select the driver.
Load Driver	Loads the specified driver onto the processor
Man. code	Specifies the flash memory's manufacturer code. Load the driver to view this data.
Device code	Displays the flash memory's device code. Load the driver to view this data.
Part description	Displays the flash memory's part description. Load the driver to view this data.
Status	Displays the utility's current status Red – The utility is not ready. Load a driver. Green – The utility is ready to process a command. Yellow – The utility is busy processing a command.
Load File	
Data file	Specifies the data file. Type the path and file name or browse to select the file. **Note:** Only valid Intel Hex files may be used. The VisualDSP++ loader produces files in this format.
Verify Flash	Compares the flash's current contents against an Intel Hex data file. This procedure assumes that a driver has been loaded.
Load File	Loads the specified data file onto the flash memory device
Global Options	
Erase all	Erases all of the device's memory
Erase based on map	Erases based on a sector map
Erase affected	Erases only the portion of flash affected by the write
No erase	Does not erase flash

Table 3-5. Flash Programmer Window Controls (Cont'd)

Control	Description
Verifying while writing flash	Verifies each performed write by ensuring that what was written matches what is read back from flash
Advanced Options	
Enable	Enables advanced features Selecting this control enables the fields and buttons below it. Clearing this control disables (grays out) the fields and buttons.
Offset	Specifies the first address at which to place data
Value	Specifies the data value to be written
Count	Specifies the number of locations to be written
Stride	Specifies the number of locations to skip between each write. Typically, this is `0x1`. Entering `0x2` specifies every other location.
Fill Flash	Loads the specified data value onto the flash memory device
Erase Sector	Erases the specified sector from the flash device
Edit Map	Opens the **Sector Map** dialog box, which enables you to select the sector (or sectors) you want to erase. This function assumes that a driver has been loaded. **Note**: Erasure is constricted by the selected Global Erase option.
Erase All	Erases the flash device's entire memory
Reset Flash	Resets the internal state of the flash device and places it into read mode without modifying its contents
Command	Specifies the custom command to be run
Send	Sends the specified custom command to the driver. The value entered in **Command** is interpreted as a hexadecimal value; for example, 10 is interpreted as 10 hexadecimal or 16 decimal.

A REFERENCE INFORMATION

This appendix contains describes VisualDSP++ features and shows you ways to speed up program development. Sections include:

- "Glossary" on page A-2
- "File Types" on page A-24
- "Toolbar Buttons" on page A-27
- "Keyboard Shortcuts" on page A-32
- "IDDE Command-Line Parameters" on page A-38
- "Extensive Scripting" on page A-39
- "Text Operations" on page A-43
- "Online Documentation" on page A-48
- "Online Help" on page A-51
- "About VisualDSP++ Dialog Box" on page A-65
- "Window Operations" on page A-70

Glossary

The following terms are important toward understanding VisualDSP++.

Application Programming Interface (API) functions

A set of functions available to an applications programmer. These functions, which are part of an application, can be accessed by other applications. For VDK, API refers to a library of C/C++ functions and assembly macros that define VDK services. These services are essential for kernel-based application programs. The services include interrupt handling, thread management, and semaphore management.

archiver

The VisualDSP++ archiver, `elfar.exe`, combines object (`.DOJ`) files into library (`.DLB`) files, which serve as reusable resources for project development. The linker searches library files for routines (library members) that are referred to by other objects, and links them in your executable program.

breakpoint

User-defined halt in an executable program. Toggle breakpoints (turn them on or off) by double-clicking on a location in a **Disassembly** window or editor window.

break condition

Hardware condition under which the target breaks and returns control of the target back to the user. For example, a break condition could be set up to occur when address `0x8000` is read from or written to.

build

Performing a build (or project build) refers to the operations (preprocessing, assembling, and linking) that VisualDSP++ performs on projects and files. During a build, VisualDSP++ processes the files in the project that have been modified (or depend on files that have been modified) since the previous build. A build differs from a rebuild all. During a rebuild all, VisualDSP++ processes all the files in the project, regardless of whether they have been modified.

build type

Replaced by "configuration"

channel

A transmission path between two communicating locations, usually the smallest subdivision of a transmission system. For VDK, channel refers to a FIFO queue into which messages sent to a thread are placed. Each thread has 15 channels. Messages are received in priority order from the lowest numbered channel to the highest.

COFF

Common Object File Format. VisualDSP++ does not support COFF formatted files.

configuration (or project configuration)

A project is developed in stages (configurations). By default, a project includes two configurations: Debug and Release.
A configuration refers to the collection of options (tool chain and individual options for files) specified for the configuration. You can add a configuration to your project at any time. You can delete a customized configuration that you created, but you cannot delete the Debug or Release configurations.

context switch

A process of saving/restoring the processor's state. The scheduler performs the context switch in response to the system change.

A hardware interrupt can occur and change the state of the system at any time. Once the processor's state has changed, the currently running thread may be swapped with a higher-priority thread. When the kernel switches threads, the entire processor's state is saved and the processor's state for the thread being switched in is restored.

critical region

A sequence of instructions whose execution cannot be interrupted or swapped out. Suspending all interrupt service routines (ISRs) before calling the critical region ensures that the execution of a critical region is not interrupted. Once the critical region routine concludes, ISRs are enabled.

CROSSCORE®

Analog Devices processor development tools, which provide easier and more robust methods for engineers to develop and optimize systems by shortening product development cycles for faster time-to-market. CROSSCORE components include the VisualDSP++ software development environment and EZ-KIT Lite evaluation systems and emulators for rapid on-chip debugging.

current directory

Directory where the `.DPJ` file is saved. The build tools use the current directory for all relative file path searches. See also "default directories."

data set

A series of data values in processor memory used as input to a plot. You can create data sets and configure the data for each data set. You specify the memory location, the number of values, and other options that identify the data. Additional specifications for row and column counts are required for 3-D plots.

Debug configuration

For a debug configuration, you can accept the default options or specify your own options and save them. The configuration refers to the specified options for all the tools in the tool chain. See also "configuration."

debug session

The combination of a target and a platform. For example, a session can be a JTAG emulator target connected to a platform consisting of five processor. Another example of a debug session is an ADSP-21262 EZ-KIT Lite target connected to an ADSP-21262 EZ-KIT Lite board.

The processor projects being developed are run as debug sessions. The two types of sessions are hardware and software. The processor, target, and platform define the session. When setting up a session, set the focus on a series of more specific elements.

debug target

The communication channel between VisualDSP++ and a processor (or group of processors). Targets include simulators, emulators, and EZ-KIT Lite evaluation systems. Several targets may be installed on your system. Simulator targets, such as the ADSP-TS101 cycle-accurate simulator, differ from emulator targets in that the processor exists only in software.

The Summit-ICE emulator communicates with one or more physical devices over the host PC's PCI bus. The Apex-ICE emulator communicates with a device through the PC's USB port.

default intermediate and output file directories

These file directories (folders) are `\Debug` (for the debug configuration) and `\Release` (for the release configuration). By default, VisualDSP++ creates these directories as children of the directory where the `.DPJ` file is saved, which is called the project's current directory. See also "current directory."

dependencies

VisualDSP++ uses dependency information to determine which files, if any, are updated during a build. If an included header file is modified, VisualDSP++ builds the source files that include (`#include`) the header file, regardless of whether the source files have been modified since the previous build.

dependency files

Usually user files or system header (`*.H`) files, these files are referenced from a source file by a preprocessor `#include` command.

device

A single processor. With regard to JTAG emulation and the JTAG EZ-ICE Configurator, a device refers to any physical chip in the JTAG chain.

device driver

A user-written model that abstracts the hardware implementation from the application code. User code accesses device drivers through a set of device driver API functions.

DWARF-2

Format for debugging source-level assembly code via improved line and symbol information

editor window

A document window that displays a source file for editing. When an editor window is active, you can move about within the window and perform typical text editing activities such as searching, replacing, copying, cutting, pasting, and so on.

ELF

Executable Linking Format

emulator

Hardware used to connect a PC to a processor target board. This hardware allows application software to be downloaded and debugged from within the VisualDSP++ environment. Emulator software performs the communications that enable you to see how your code affects processor performance.

event

A signal (similar to a semaphore or message) used to synchronize multiple threads in a system. An event is a logical switch, having two binary states (available/true and unavailable/false) that control thread execution. When an event becomes available, all pending (waiting) threads in the wait list are set to a ready-to-run state. When an event is available and a thread pends on it, the thread continues running and the event remains available.

To facilitate error handling, threads can specify a timeout period when pending on an event.

An event is a code object of global scope, so any thread can pend on any event. Event properties include the `EventBit` mask, `EventBit` value, and combination type. Events are statically allocated and enumerated at run time. An event cannot be destroyed, but its properties can be changed.

event bit

A flag set or cleared to post the event. The event is posted (available) when the current values of the system Event Bits match the event bit's mask and event bits' values defined by the event's combination type.

A system has only one Event Bits word, the size of a data word minus one. For ADSP-TSxxx processors, the size is 31 bits.

executable file

A file or program written and built in VisualDSP++

EZ-KIT Lite evaluation system

A development board, software, and cable for evaluating a particular processor. The kit includes fundamental debugging software to facilitate architecture evaluations via a PC-hosted tool set. Use the kit to evaluate Analog Devices processors, learn about processor applications, simulate and debug applications, and prototype applications.

focus

Refers to the active processor in a multiprocessor (MP) debugging session

ICE

In-Circuit Emulator. Analog Devices offers emulators that provide non-intrusive target-based debugging of processor systems. An emulator can single-step or execute a processor at full speed to facilitate viewing or altering a processor's register and memory contents.

IDDE

Integrated Development and Debugging Environment for Analog Devices processor development tools

interrupt

An external or internal condition detected by the hardware interrupt controller. In response to an interrupt, the kernel processes a subroutine call to a predefined interrupt service routine (ISR).

Interrupts have the following specifications.

Latency – interrupt disable time. The period between the interrupt occurrence and the first ISR's executed instruction.

Response – interrupt response time. The period between the interrupt occurrence and a context switch.

Recovery – interrupt recovery time. The period needed to restore the processor's context and to start the return-from-interrupt (RTI) routine.

interrupt service routine (ISR)

A routine executed as a response to a software interrupt or hardware interrupt. VDK supports nested interrupts, which means that the kernel recognizes other interrupts, services interrupts, or both with higher priorities while executing the current ISR. For VDK, the ISRs are written in assembly language. VDK reserves the timer and the lowest priority (reschedule) interrupt.

JTAG

Joint Test Action Group. This committee is responsible for implementing the IEEE boundary scan specification, enabling in-circuit emulation of ICs.

JTAG ICE configurator

See "VisualDSP++ configurator."

kernel

The main module of a real-time operating system. The kernel loads first and permanently resides in the main memory and manages other modules of the real-time operating system. Typical services include context switching and communication management between OS modules.

keyboard shortcuts

The keyboard provides a quick means of running the commands used most often, such as simultaneously typing the keyboard's **Ctrl** and **G** keys (indicated with the symbols **Ctrl+G**) to go to a line in a file.

librarian

A utility that groups object files into library files. When linking your program, specify a library file and the linker automatically links any file in the library that contains a label used in your program. Source code is provided so you can adapt the routines to your needs.

library files

The VisualDSP++ archiver, `elfar.ex`, combines object (`.DOJ`) files into library (`.DLB`) files, which serve as reusable resources for project development. The linker searches library files for routines (library members) that are referred to from other objects, and links them into your executable program.

linear profiling

A debugging feature that samples the target's PC register at every instruction cycle. Linear profiling gives an accurate picture of where instructions were executed, since every PC value is collected. The trade-off, however, is that linear profiling is much slower than statistical profiling. A display of the resulting samples appears in the **Linear Profiling Results** window, which graphically indicates where the application is spending its time. Simulator targets support linear profiling. See also "Statistical profiling."

linker

The linker creates executable files, shared memory files, and overlay files from separately assembled object and library files. It assigns memory locations to code and data in accordance with a user-defined `.LDF` file, which describes the memory configuration of the target system.

Linker Description Files (.LDF files)

The `.LDF` files describe the target system and map your program code within the system memory and processors. The `.LDF` file creates an executable file using the target system memory map and defined segments in your source files.

loader

A utility that transforms an executable file into a boot file. The loader creates a small kernel, which is booted into internal memory at chip reset. A program of arbitrary size can then be loaded into the processor's internal and external memory.

makefile

VisualDSP++ can export a makefile (make rule file), based on your project options. Use a makefile (`.MAK` or `.MK`) to automate builds outside of VisualDSP++. The output make rule is compatible with the gnumake utility (GNU Make V3.77 or higher) or other make utilities.

memory pool

An area of memory containing a specified number of uniformly sized blocks of memory available for allocation and subsequent use in an application. The number and size of the blocks in a particular memory pool are defined at pool creation.

message

For VDK, a signal (similar to an event or semaphore) used to synchronize two threads in a system or to communicate information between threads. A message is sent to a specified channel on the recipient thread (and can optionally pass a reference to a payload to facilitate the transfer of data between threads). Posting a message takes a deterministic amount of time and may incur a context switch.

mixed mode

One of the two editor window display formats (the other being source mode). Mixed mode displays assembled code after the line of the corresponding C code.

multiprocessor system

A system built with multiple processors. Often, performance-based products require two or more processors. A system built with a single processor is called a *single-processor system*. Debugging a multiprocessor system requires that you synchronously run, step, halt, and observe program execution operations in all the processors at once. The SHARC and TigerSHARC simulators do not support this capability.

non-bootable PROM-image file

Splitter output, consisting of PROM files that cannot be used to boot-load a system

outdated file

A file that has been edited since the last build

payload

For VDK, an arbitrary amount of data associated with a message. A reference to the payload can be passed between threads as part of a message to enable the recipient thread to access the data buffer that contains the payload.

pinning a window

A technique that statically associates a window to a specific processor

pipelining

A feature that helps you analyze and tune your code for optimal performance. For TigerSHARC processors and Blackfin processors, VisualDSP++ provides a simulation-only debugging window (**Pipeline Viewer**) to help visualize the pipeline by displaying pipeline stalls and aborts. For SHARC processors, the **Disassembly** window displays symbols (F, D, or E) to indicate an instruction's pipeline stage.

platform

The device with which a target communicates. For simulation, a platform is typically one or more processors of the same type. For emulation, you specify the platform with the VisualDSP++ configurator, and the platform can be any combination of devices.

The platform represents the hardware upon which one or more devices reside. You typically define a platform for a particular target. For example, if three emulators are installed on your system, a platform selection might be emulator two.

Several platforms may exist for a given debug target. For a simulator, the platform defaults to the identical processor simulator. When the debug target is a JTAG emulator, the platforms are the individual JTAG chains. When the debug target is an EZ-KIT Lite board, the platform is the board in the system on which you wish to focus.

pre-emptive kernel

A priority-based kernel in which the currently running thread of the highest priority is pre-empted, or suspended, to give system resources to the new highest-priority thread

processor

An individual chip contained on a specific platform within a target system. When you create the executable file, the processor is specified in the Linker Description File (`.LDF` file) and other source files.

profile-guided optimization (PGO)

A process that involves setting up and executing data sets to produce an optimized application. A *data set* is the association of zero or more input streams with one `.PGO` output file.

profiling

A technique used during simulation to examine program execution within selected ranges of code. Profiling helps you determine: percentage of time spent executing instructions, number of clock cycles spent executing instructions, number of instructions executed, and the number of times memory is read or written.

The profiler is non-intrusive. It does not report on execution within a called function ("daughter" function). Use profiling to monitor program memory. By watching one or more profile ranges, you can find areas of code that may be optimized for better performance. A profile session must include one memory range at a minimum. For each range, specify a start and end address. You can use symbols or hexadecimal numbers to represent addresses.

project

This term refers to the collection of source files and tool configurations used to create a processor program. Through a project, you can add source files, define dependencies, and specify build options related to producing your output executable program. A project (`.DPJ`) file stores your program's build information.

VisualDSP++ helps you manage projects from start to finish in an integrated user interface. Within the context of a processor project, you define project and tool configurations, specify project-wide and individual file options for debug or release modes of project builds, and create source files. VisualDSP++ facilitates easy movement among editing, building, and debugging activities.

project configuration

This configuration includes all of the settings (options) for the tools used to build a project.

project file tree display

See “Project window.”

Project window

This window displays your project’s files in a *tree view*, which can include folders to organize your project files. Right-clicking on an icon (the project itself, a folder, or a file) opens a menu of actions that you can perform on the selected item. Double-clicking on the project icon or a folder icon opens or closes the tree list. Double-clicking a file icon opens the file in an editor window

Project wizard

Simplifies the creation of a new project by opening a series of pages from which to specify options. For Blackfin processors, additional pages facilitate the inclusion of startup code. You can modify project options at a later time via the **Project Options** dialog box.

real-time operating system (RTOS)

A software executive that handles processor algorithms, peripherals, and control logic. The RTOS comprises these components: kernel, communication manager, support library, and device drivers. An RTOS enables structured, scalable, and expandable processor application development while hiding OS complexity.

rebuild all

See “build.”

registers

For information on available registers, see the corresponding processor documentation or view the associated online Help.

release configuration

You can accept the default set of options, or you can specify the options you want and save them. The configuration refers to the specified options for all the tools in the tool chain. See also "Configuration."

reset

This command resets the processor to a known state and clears processor memory.

restart

This command sets your program to the first address of the interrupt vector table. Unlike a reset, a restart does not reload memory.

right-click

This action opens a right-click menu (sometimes called a context menu, pop-up menu, or shortcut menu). The commands that appear depend on the context (what you are doing). Right-click menus provide access to many commonly used commands.

round-robin scheduling

For VDK, a scheduling scheme whereby all threads at a given priority are given processor time automatically in fixed duration intervals. Round-robin priorities are specified at build time.

scheduler

For VDK, a kernel component responsible for scheduling system threads and interrupt service routines. VDK is a priority-based kernel in which the highest-priority thread is executed first.

scripting

You can interact with the IDDE by using a single command or a script file. Scripting languages include VBScript, JavaScript, and Tcl. Output displays in the **Console** view of the **Output** window. The output is also logged to the `VisualDSP_log.txt` file.

semaphore

For VDK, a signal (similar to an event or message) used to synchronize multiple threads in a system. A semaphore is a data object whose value is zero or a positive integer (limited by the maximum setup at creation time). The two states (available/greater than zero and unavailable/zero) control thread execution. Unlike an event, whose state is automatically calculated, a semaphore is directly manipulated. Posting a semaphore takes a deterministic amount of time and may incur a context switch.

serial port data

You can automatically transfer serial port (SPORT) data to and from on-chip memory by using DMA block transfers. Each serial port offers a time division multiplexed (TDM) multichannel mode.

session

See "debug session."

session name

Although the choice of target, platform, and processor define the session, you may want to further identify the session. To prevent confusion later, modify the default session name when you first create the debug session. A session name can be any string and can include space characters. There is no limit to the number of characters in a session name, but the **Session List** dialog box can display about 32 characters.

shortcuts

See “keyboard shortcuts.”

signal

For VDK, a method of communicating between multiple threads. VDK supports four types of signals: semaphores, events, messages, and device flags.

simulator

The simulator is software that mimics the behavior of a processor chip. Simulators are often used to test and debug code before the processor chip is manufactured.

The way a simulator runs an executable program in software is similar to the way a processor does in hardware. The simulator also simulates the memory and I/O devices specified in the `.LDF` file. VisualDSP++ lets you interactively observe and alter the data in the processor and in memory. The simulator reads executable files. A simulator’s response time is slower than that of an emulator.

source files

The C/C++ language and assembly language files that make up your project. Other source files that a project uses, such as the `.LDF` file, contain command input for the linker and dependency files (data files and header files). View source files in editor windows.

source mode

One of the two editor window display formats (the other being mixed mode). Source mode displays C code only.

splitter

A PROM splitter utility that transforms an executable file into a non-boot-loadable image. This file is loaded onto external processor memory.

statistical profiling

A debugging feature that provides a more generalized form of profiling that is well suited to JTAG emulator debug targets. With statistical profiling, VisualDSP++ randomly samples the target processor's program counter (PC) and presents a graphical display of the resulting samples in the **Statistical Profiling Results** window. This window graphically indicates where the application is spending time.

JTAG sampling is completely non-intrusive, so the process does not incur additional run-time overhead. See also "linear profiling."

stepping

A technique for moving through source or assembly code to observe instruction execution

streams

A debug tool used during simulation to drive other devices or take part in processing a subset of data. Use streams to simulate data input and output.

symbols

Labels for sections, subroutines, variables, data buffers, constants, or port names. For more information, refer to the related build tool documentation.

system configurator

For VDK, the system configuration control is accessible from the **Kernel** page of the **Project** window. The **Kernel** page provides a graphical representation of the data contained in the `vdk.h` and `vdk.cpp` files.

target

See "Debug target."

threads

For VDK, a kernel system component that performs a predetermined function and has its own share of system resources. VDK supports multithreading, a run-time environment with concurrently executed independent threads.

Threads are dynamic objects that can be created and destroyed at runtime. Thread objects can be implemented in C, C++, or assembly language. A thread's properties include an ID, priority, and current state (wait, ready, run, or interrupted). Each thread maintains its own C/C++ stack.

ticks

The system-level timing mechanism. Every system tick is a timer interrupt.

tool chain

The collection of tools (utilities) used to build a project configuration

trace

Provides a history of program execution. A trace is sometimes called an execution trace or a program trace. Trace results show how the program arrived at a certain point and show program reads, writes, and memory fetches. Blackfin and TigerSHARC processors do not support traces.

unscheduled regions

For VDK, a sequence of instructions whose execution can be interrupted, but cannot be swapped out. The kernel acknowledges and services interrupts when an unscheduled region routine is running.

VDK

See "VisualDSP++ Kernel (VDK)."

VisualDSP++

An Integrated Development and Debugging Environment (IDDE) for Analog Devices processor development tools

VisualDSP++ Configurator

Previously called JTAG ICE Configurator, use this utility to describe the hardware to VisualDSP++ when connecting to a JTAG emulator session. VisualDSP++ requires this description to set up the debug session. The VisualDSP++ Configurator also provides access to ICE Test, a utility for testing the target.

VisualDSP++ Kernel (VDK)

The RTOS kernel from Analog Devices, a software executive between processor algorithms, peripherals, and control logic. The kernel is integrated with the Integrated Development and Debugging Environment (IDDE), assembler, compiler, and linker programs into the development tool chain.

Refer to the *VisualDSP++ 4.0 Kernel (VDK) User's Guide* for details.

watchpoints

For simulation only. Similar to breakpoints, watchpoints stop program execution. Unlike breakpoints, which are attached to specific addresses, watchpoints are attached to user-defined conditions, such as memory reads or stack pops. The program halts when the conditions are met. SHARC processors do not support watchpoints.

workspace

You can open multiple windows, arrange them in any configuration, and save the layout as a workspace setting that can be recalled (loaded) at a later time. Each debug session's default workspace is automatically saved when you close the session and is automatically restored when you load that session.

By default, each session includes two workspaces. You can create any number of workspaces and switch among them. Suggested workspaces include an edit workspace, a debug workspace, and a plot workspace.

File Types

Table A-1 describes processor project files used by VisualDSP++.

Table A-1. Files Used With VisualDSP++

Extension	Name	Purpose
.ASM	Assembly source file	Source file comprising assembly language instructions
.C	C source file	Source file comprising ANSI standard C code and Analog Devices extensions
.CPP .CXX .HPP .HXX	C++ source file	Preprocessed compiler files that are inputs to the C/C++ compiler. These files comprise ANSI standard C++ code.
.DPJ	Project file	Contains a description of how your source files combine to build an executable program
.LDF	Linker Description File	Linker command source file is a text file that contains commands for the linker in the linker's scripting language
.IS .PP .S	Intermediate files	Preprocessed assembly files generated by the preprocessor
.DOJ	Assembler object file	Binary output of the assembler
.DLB	Archiver file	Archiver's binary output in ELF format
.H	Header file	Dependency file used by the preprocessor, and a source file for the assembler and compiler
.DAT	Data file	Dependency file used by the assembler for data initialization
.DLO .DXE .OVL .SM	Debugging files	Binary output files from the linker in ELF/DWARF format

Table A-1. Files Used With VisualDSP++ (Cont'd)

Extension	Name	Purpose
.MAP	Linker memory map file	Optional output for the linker. This text file contains memory and symbol information for executable files.
.TCL .TC8	Tool Command Language files	Tcl scripting language files used to script work
.OBJ	Assembled object file	(Previous releases only, replaced by .DOJ) Output of the assembler
.LST	Listing file	Optional file output by the assembler
.BNM .H .LDR	Loader format files	The loader's output in ASCII format. Different varieties exist. Used to create boot PROMS.
.H_# .S_# .STK	PROM format files	The loader's output in ASCII format. Different varieties exist. Used to create boot PROMS.
.ACH	Architecture file	(Previous releases only, replaced by .LDF)
.TXT	Linker command-line file	(Previous releases only, replaced by .LDF) ASCII text file that contains command-line input for the linker
.EXE	Debugging file	(Used in previous releases, replaced by .DXE)
.EXE	Compiled simulation file	Enables faster execution speed compared to a standard .DXE program
.VDK	VisualDSP++ kernel support file	Enables VDK support
.JS .VBS	Script files	Enable you to script work for test applications. Scripting languages let you access the Automation API to interact with the IDDE.
.processor	Assembly source file	Source file comprising assembly language instructions

Table A-1. Files Used With VisualDSP++ (Cont'd)

Extension	Name	Purpose
.MAK .MK	Makefiles	The output make rule file is used for project builds
.DPG	Project group	An .XML file containing information about projects

Toolbar Buttons

The toolbar, which comprises separate tool buttons, provides quick mouse access to commands.

The toolbar is a Windows docking bar. You can move it to different areas of the screen by dragging it to the selected location.

Table A-2. Toolbar Buttons

Button	Purpose
	Creates a new document
	Opens an existing document
	Saves the active document or template with the same name
	Saves all open files that have been modified, including files not in the current project
	Prints the active document
	Loads a program into the target
	Reloads the most recent program into the target
	Cuts selected data from the document and store it on the clipboard
	Copies the selection to the clipboard
	Pastes the contents of the clipboard at the insertion point

Table A-2. Toolbar Buttons (Cont'd)

Button	Purpose
	Undoes previous edit command (multilevel undo)
	Redoes the command undone by the previous Undo command (multilevel redo)
	Finds a text block in an editor window
	Finds again or repeats the previous find command
	Replaces the selected text with other text
	Searches through files for text or regular expressions
	Goes to or moves to the specified location
	Displays the current source file
	Toggles the bookmark at selected line in the active editor window
	Goes to the next bookmarked line in the editor window
	Goes to the previous bookmarked line in the editor window
	Clears all bookmarks in the editor window
	Opens the online Help to the **Search** page

Table A-2. Toolbar Buttons (Cont'd)

Button	Purpose
	Provides context-sensitive Help for a button command or portion of VisualDSP++
	Opens the **About VisualDSP++** dialog box
	Adds a source file to the project
	Removes a selection from the project
	Opens an existing project
	Saves the open project
	Opens the **Project Options** dialog box, where you specify project options
	Builds the selected source file
	Builds the project (update outdated files)
	Builds all files in the project
	Stops the current project build
	Arranges windows as tall non-overlapping tiles
	Arranges windows as wide non-overlapping tiles

Table A-2. Toolbar Buttons (Cont'd)

Button	Purpose
	Arranges windows so they overlap
	Closes all open windows
	Refreshes all the debugging windows
	Runs (starts or continues) the current program
	Restarts the current program
	Stops the current program
	Resets the target
	Toggles a breakpoint for the current line
	Clears all current breakpoints
	Enables or disables one breakpoint
	Disables all breakpoints
	Steps one line
	Steps over the current statement

Table A-2. Toolbar Buttons (Cont'd)

Button	Purpose
	Steps out of the current function
	Runs the program to the line containing the cursor
	Opens the **Expressions** window
	Opens the **Locals** window
	Opens the **Call Stack** window
	Opens the **Disassembly** window
	Runs the command associated with the user tool (one of ten)
	Opens the associated workspace (one of ten)

Keyboard Shortcuts

VisualDSP++ includes keyboard shortcuts (also called shortcut keys) for commonly used operations. These keyboard shortcuts appear in the tables below. You can also access and run commands by:

- Clicking on menu items (and commands) in the menu bar
- Clicking toolbar buttons
- Right-clicking from a particular context, such as from the **Project** window
- Clicking configured user tools (for example,)
- Clicking buttons in dialog boxes
- Running scripts (via the **File** menu, **Output** window, or editor window)
- Choosing a command from a control menu

Working With Files

When working with files, use the keyboard shortcuts listed in Table A-3.

Table A-3. Keyboard Shortcuts for Working With Files

Action	Key(s)
Open a new file	**Ctrl+N**
Open an existing file	**Ctrl+O**
Save a file	**Ctrl+S**
Print a file	**Ctrl+P**
Go to the next window	**F6**
Go to the previous window	**Shift+F6**

Moving Within a File

To move within a file, use the keyboard shortcuts listed in Table A-4.

Table A-4. Keyboard Shortcuts for Moving Within a File

Action	Key(s)
Move the cursor to the left one character	Left Arrow (←)
Move the cursor to the right one character	Right Arrow (→)
Move the cursor to the beginning of the file	**Ctrl+Home**
Move the cursor to the end of the file	**Ctrl+End**
Move the cursor to the beginning of the line	**Home**
Move the cursor to the end of the line	**End**
Move the cursor down one line	Down Arrow (↓)
Move the cursor up one line	Up Arrow (↑)
Move the cursor one page down	**Page Down**
Move the cursor one page up	**Page Up**
Move the cursor right one tab	**Shift**
Move the cursor left one tab	**Shift+Tab**
Move the cursor left one word	**Ctrl**+Left Arrow (←)
Move the cursor right one word	**Ctrl**+Right Arrow (→)
Move to the matching brace character within a file	**Ctrl+B**
Go to the next bookmark	**F2**
Go to a line	**Ctrl+G**
Find text	**Ctrl**+F
Find the next occurrence of text	**F3**

Cutting, Copying, Pasting, Moving Text

To edit text, use the keyboard shortcuts listed in Table A-5.

Table A-5. Keyboard Shortcuts for Editing Text

Action	Key(s)
Copy text	**Ctrl+C** or **Ctrl+Insert**
Copy text	Select with cursor and **Ctrl**+drag
Cut text	**Ctrl+X** or **Shift+Delete**
Delete text	**Delete** (selection or forward)
Delete text	**Backspace** (selection or backward)
Move text	Select with cursor and drag
Move selected text right one tab	**Tab**
Move selected text left one tab	**Shift+Tab**
Paste text	**Ctrl+V** or **Shift+Insert**
Undo the last edit	**Ctrl+Z** or **Alt+Backspace**
Redo an edit command	**Shift+Ctrl+Z**
Replace text	**Ctrl+H** or **Ctrl+R**

Selecting Text Within a File

To select text within a file, use the keyboard shortcuts listed in Table A-6.

Table A-6. Keyboard Shortcuts for Selecting Text Within a File

Action	Key(s)
Select all text in a file	**Ctrl+A**
Select the character on the left	**Shift**+Left Arrow (←)
Select the character on the right	**Shift**+Right Arrow (→)

Table A-6. Keyboard Shortcuts for Selecting Text Within a File (Cont'd)

Action	Key(s)
Select all text to the beginning of the file	**Shift+Ctrl+Home**
Select all text to the end of the file	**Shift+Ctrl+End**
Select all text to the beginning of the line	**Shift+Home**
Select all text to the end of the line	**Shift+End**
Select all text to the line below	**Shift**+Down Arrow (↓)
Select all text to the line above	**Shift**+Up Arrow (↑)
Select all text to the next page	**Shift+PgDn**
Select all text to the above page	**Shift+PgUp**
Select the word on the left	**Shift+Ctrl**+Left Arrow (←)
Select the word on the right	**Shift+Ctrl**+Right Arrow (→)
Select by column	Place cursor, press and hold down **Alt** and drag the cursor (selects by column-character instead of by line-character)

Working With Bookmarks in an Editor Window

When working with bookmarks in an editor window, use the keyboard shortcuts listed in Table A-7.

Table A-7. Keyboard Shortcuts for Bookmarks

Action	Key(s)
Toggle a bookmark	**Ctrl+F2**
Go to next bookmark	**F2**

Building Projects

To build projects, use the keyboard shortcuts listed in Table A-8.

Table A-8. Keyboard Shortcuts for Building Projects

Action	Key(s)
Build the current project	F7
Build only the current source file	Ctrl+F7

Using Keyboard Shortcuts for Program Execution

For program execution, use the keyboard shortcuts listed in Table A-9.

Table A-9. Keyboard Shortcuts for Program Execution

Action	Key(s)
Load a Program	Ctrl+L
Reload a Program	Ctrl+R
Dump to File	Ctrl+D
Run	F5
Multiprocessor Run	Ctrl+F5
Run to Cursor	Ctrl+F10
Halt	Shift+F5
Step Over	F10
Step Into	F11
Multiprocessor Step	Ctrl+F11
Step Out Of	Alt+F11
Halt a Script	Ctrl+H

Working With Breakpoints

When working with breakpoints, use the keyboard shortcuts listed in Table A-10.

Table A-10. Keyboard Shortcuts for Breakpoints

Action	Key(s)
Open the **Breakpoints** dialog box	**Alt+F9**
Enable/disable a breakpoint	**Ctrl+F9**
Toggle (add or remove) a breakpoint	**F9**

Obtaining Online Help

To obtain online Help, use the keyboard shortcuts listed in Table A-11.

Table A-11. Keyboard Shortcuts for Obtaining Online Help

Action	Key(s)
View online Help for the selected object	**F1**
Obtain context-sensitive Help for controls (buttons, fields, menu items)	**Shift+F1**

Miscellaneous

For windows and workspaces, use the keyboard shortcuts listed in Table A-12.

Table A-12. Miscellaneous Keyboard Shortcuts

Action	Key(s)
Refresh all windows	**F12**
Select workspace 1 through 10	**Alt+1 … Alt+0**

IDDE Command-Line Parameters

VisualDSP++ can be invoked from a command line or a shortcut.

Syntax:

```
idde.exe [-f script_name]
         [-s session_name]
         [-p project_name]
```

Note: Specify the full path to `idde.exe`. Only one instance of each parameter is permitted.

Table A-13 describes the `idde.exe` command-line parameters.

Table A-13. idde.exe Command-Line Parameters

Item	Description
`-f script_name`	Loads and executes the script specified by `script_name`. Use this parameter to automate regression tests. You can also manipulate VisualDSP++ by running a script from a library of common commands that you create. If an error is encountered while executing this script, VisualDSP++ automatically exits.
`-s session_name`	Specifies the session to which VisualDSP++ connects when it starts. The session must already exist. This parameter is useful when you are debugging more than one target board. Having multiple shortcuts to `idde.exe` allows you to run a different session. This overrides VisualDSP++'s default behavior of always connecting to the last session.
`-p project_name`	Specifies the project to load at startup. The project must already exist.

Examples:

```
idde.exe -f "c:\\scripts\\myscript.vbs"

idde.exe -s "My BF535 JTAG Emulator Session"

idde.exe -p "c:\\projects\\myproject.dpj"
```

Extensive Scripting

Issue script commands from a command window, the **Output** window's **Console** view, from a menu, from an editor window, or from a user tool. Refer to "Script Command Output" on page 2-48 for details on scripting.

- Command window issuance

  Load a script from a command window with an `idde` command by typing:

  ```
  idde -f script_filename
  ```

  Optionally, add `-s` and the session name to specify a previously created session. If no session name is specified, the last session is used.

  If the script encounters an error during execution, VisualDSP++ automatically exits.

- **Output** window issuance

  Load a script from the **Output** window's **Console** view by typing one of the following commands.

  For the Microsoft ActiveX script engine, type:

  ```
  Idde.LoadScript script_filename
  ```

  For Tcl, type:

  ```
  source filename
  ```

  Similar to C/C++, use a backslash (`\`) as an escape character. If you specify paths in the Windows environment, you must escape the escape character, as shown in this example:

  ```
  c:\\my_dir\\my_subdir\\my_file.vbs
  ```

For Tcl only, you may also use forward slashes to delimit directories in a path, as shown in this example:

```
source c:/my_dir/my_subdir/my_file.tcl
```

Command execution is deferred until a line is typed without a trailing backslash. This permits the entry of an entire block of code (or entire procedure) for the script interpreter to evaluate at once.

Use the built-in `Idde` object to easily access the properties and methods of the VisualDSP++ Automation API when using a Microsoft ActiveX script engine. For example:

```
Idde.ActiveSession.ActiveProcessor
```

Evaluate expressions by using the "`?`" when a Microsoft ActiveX script engine is selected. For example:

```
? Idde.FullName
```

- Menu issuance

  You can quickly issue frequently used scripts. From the **File** menu, choose **Recent Scripts** and then select the script.

- Editor window issuance

  In an open editor window that contains a script, right-click and choose **Load Script**, as shown in Figure A-1.

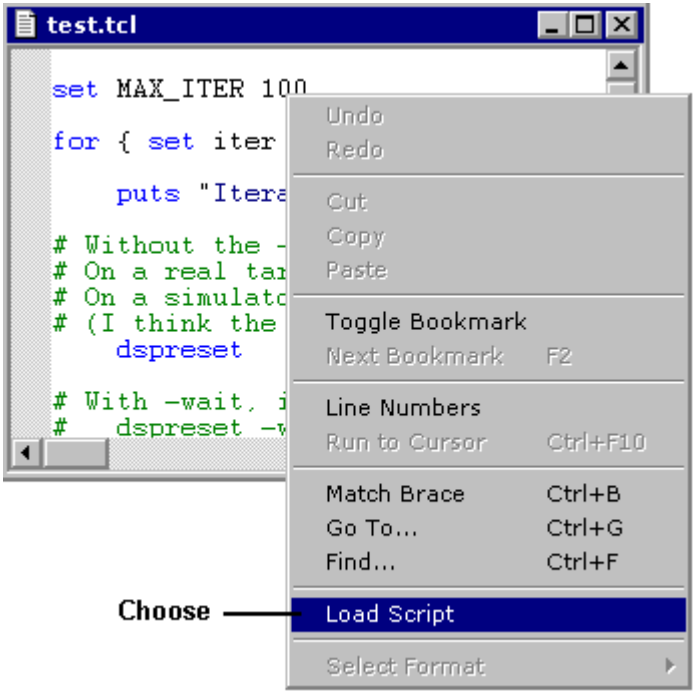

Figure A-1. Running a Script in an Editor Window

- User tool issuance

  From the User Tools toolbar, click a user tool. Alternatively, choose a user tool from the **Tools** menu.

You can invoke a script (such as `.JS` or `.VBS`) automatically when launching VisualDSP++ from a shortcut on your Window's desktop or **Start** button. Right-click on the shortcut and select **Properties** and the **Shortcut** tab. Then append `-f` and the name of the script file to the executable file in the **Target** text box.

The example shown in Figure A-2 runs `myscript.js` automatically when `idde.exe` is launched.

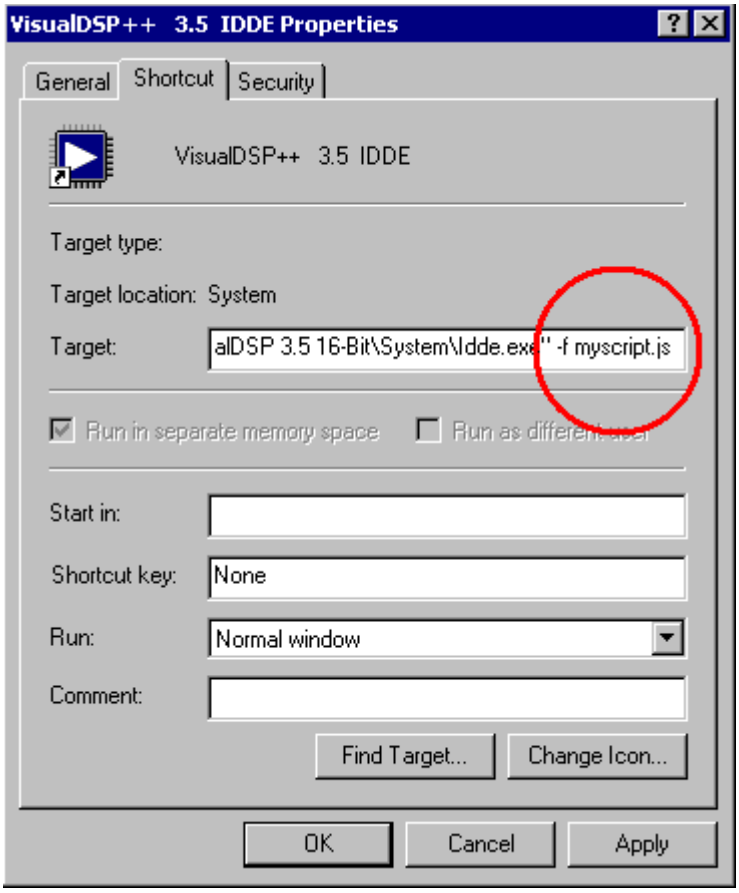

Figure A-2. Loading a Script From a Shortcut

Text Operations

VisualDSP++ allows the use of regular expressions and tagged expressions in find/replace operations and comments in your code.

Regular Expressions vs. Normal Searches

Normally, when you search for text, the search mechanism scans for an exact, character-by-character match of the search string, which does not have to be an entire word. Every character in the search string is examined. If there are embedded spaces, for instance, the exact number is matched.

Regular expression matching provides much more flexibility and power than a normal search. A regular expression can be a simple string, which yields the same matches as normal searches. Some characters in a regular expression string, however, have special interpretations, which provide greater flexibility.

For example, with regular expression matching, you can find the following.

- All occurrences of either `hot` or `cold`
- Occurrences of `for` followed by a left parenthesis, with any number of intervening spaces
- A semicolon (`;`) only when it is the last character on a line
- The string `ADSP` followed by a sequence of digits

Using a regular expression as the search pattern for replacement provides ways to identify and recover the variable portions of the matched strings.

Specific Special Characters

Regular expressions assign special meaning to the following characters.

If you have to match on one of these characters, you must escape it by preceding it with a backslash (\). Thus, `\^` matches the ^ character, yet ^ matches the beginning of the line.

Table A-14. Special Search Characters

Character	Description
^	A caret matches the beginning of the line.
$	A dollar sign matches the end of the line.
.	A period (.) matches any character.
[abc]	A bracketed sequence of characters matches one character, which may be any of the characters inside the brackets. Thus, `[abc]` matches an a, b, or c.
[0–9]	This shorthand form is valid within the sequence brackets. It specifies a range of characters, from first through last, exactly as if they had been written explicitly. Ranges may be combined with explicit single characters and other ranges within the sequence. Thus, `[-+.0-9]` matches any constituent character of a signed decimal number; and `[a-zA-Z0-9_]` matches a valid identifier character, either lowercase or uppercase. Ranges follow the ordering of the ASCII character set.
[^abc] [^0–9]	A caret (^) that is the first character of a sequence matches all characters except for the characters specified after the caret.
(*material*)	The material inside the parentheses can be any regular expression. It is treated as a unit, which can be used in combination with other expressions. Parenthesized material is also assigned a numerical tag, which may be referenced by a replace operation.

Special Rules for Sequences

The normal special character rules of regular expressions do not apply within a bracketed sequence. Thus, `[*&]` matches an asterisk or ampersand.

Certain characters have special meaning within a sequence. These include `^` (not), `-` (range), and `]` (end of sequence). By placing these characters appropriately, you can specify these characters to be part of the sequence.

To search for a right bracket character, place `]` as the first character of the search string. To search for a hyphen character, place `-` as the first character of the search string after `]`, if present. Place a caret anywhere in the search string except at the front, where it means "not."

Repetition and Combination Characters

Each character described in Table A-15 extends the meaning of the item that immediately precedes it. This item may be a single character, a sequence in braces, or an entire regular expression in parentheses.

Table A-15. Match Characters

Character	Description
*	An asterisk matches the preceding any number of times, including none at all. Thus, `ap*le` matches `apple`, `aple`, `apppppple` and `ale`. For example, `^ *void` matches only when `void` occurs at the beginning of a line and is preceded by zero or more spaces.
+	A plus character matches the preceding any number of times, but at least one time. Thus, `ap+le` matches `apple` and `aple`, but does not match `ale`.
?	A question mark matches the preceding either zero or one time, but not more. Thus, `ap?le` matches `ale` and `aple`, but nothing else.
|	The pipe character (|) matches either the preceding or following item. For example, `(hot)|(cold)` matches either `hot` or `cold`. Spaces are characters. Thus, `(hot)	(cold)` matches "`hot` "or" `cold`".

Match Rules

If multiple matches are possible, the `*`, `+`, and `?` characters match the longest candidates. The `|` character matches the left-hand alternative first.

For more information, see the many reference texts available on this topic, such as *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools* by Jeffrey E. F. Friedl, (c) 1997 O'Reilly & Associates, Inc.

Tagged Expressions in Replace Operations

Use a tagged expression as part of the string in the **Replace** field for a replace operation.

A tagged expression must be enclosed between parentheses characters. In the **Replace** field, the operators in Table A-16 represent tagged expressions from the **Find** field.

Table A-16. Using Tagged Expressions in Replace Operations

Find Field	Replace Field
Entire matched substring	`\0`
Tagged expressions within parentheses `()` from left to right	`\1 \2 \3 \4 \5 \6 \7 \8 \9`
Entire match expression	`&`

The replace expression can specify an ampersand (`&`) character, meaning that the `&` represents the substring that was found. For example, if the substring that matched the regular expression is "`abcd`", a replace expression of "`xyz&xyz`" changes it to "`xyzabcdxyz`". The replace expression can also be expressed as "`xyz\0xyz`", where the "`\0`" indicates a tagged expression representing the entire matched substring. Similarly, you can have another tagged expression represented by "`\1`", "`\2`".

Although the tagged expression 0 is always defined, the tagged expressions 1, 2, and so on, are defined only when the regular expression used in the search has enough sets of parenthesis. Some examples are shown in Table A-17.

Table A-17. Examples of Replace Operations

String	Search	Replace	Result
Mr.	(Mr)(\.)	\1s\2	Mrs.
abc	(a)b(c)	&-\1-\2	abc-a-c
bcd	(a|b)c*d	&-\1	bcd-b
abcde	(.*)c(.*)	&-\1-\2	abcde-ab-de
cde	(ab|cd)e	&-\1	cde-cd

Comment Start and Stop Strings

Use start comment strings and stop comment strings for comment highlighting colors. Table A-18 describes the two types of comment strings that you can set for each file type.

Table A-18. Start and Stop Comment Strings

String	Purpose
!	Starts an assembly style, single-line comment
/*	Starts a C/C++ style, multiline comment
//	Starts a C/C++ style, single-line comment
Carriage return	Ends a single-line comment (C and Assembly)
*/	Ends a C/C++ style, multiline comment
(blank)	Ends a C/C++ style, single-line comment

Online Documentation

VisualDSP++ includes three types of user documentation: Help files, PDF files, and HTML files.

Table A-19. Types of User Documentation

Files	Purpose
.CHM	VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the <installation>\Help folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ **Help** menu or via the Windows **Start** button. The .CHM files require Internet Explorer 4.0 (or higher) or the installation of a component that provides a .CHM file viewer.
.PDF	Manuals and data sheets in Portable Documentation Format are located in the installation CD's Docs folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running setup.exe on the installation CD provides easy access to these documents. You can also copy PDF files from the installation CD onto another disk.
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the Docs\Reference folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.

The VisualDSP++ software installation procedure does not copy PDF versions of books and data sheets or supplemental reference documentation to the VisualDSP++ installation directory.

Printing Online Documentation

Besides printing topics from Help (on page A-48), you can print large documents (VisualDSP+ manuals, hardware manuals and data sheets, and more) from the VisualDSP++ Tools Installation CD-ROM.

To print documents:

1. Insert the VisualDSP++ Tools Installation CD-ROM in the CD-ROM drive.
2. Open the **Docs** folder by using one of these options:
   - From the **VisualDSP++ Tools Installation** main menu, click **View Documentation**. (If the main menu does not appear, run `setup.exe`.)
   - In Windows Explorer, select the CD-ROM drive (for example, **D:**) and open the **Docs** folder.
3. Open the folder where the document is located.

   The `Data Sheets` folder contains processor data sheets. Be sure to check the Analog Devices Web site for updated versions.

   The `Hardware Manuals` folder contains hardware manuals.

   The `Reference` folder includes the `.HTML` files that comprise the Dinkum Abridged C++ library and the FlexLM network license documentation.

   The `Tools Manuals` folder contains VisualDSP++ tools manuals.
4. Double-click the document that you want to print. Selecting a `.PDF` file opens Adobe Acrobat Reader and displays the document. Selecting an `.HTML` file opens a browser and displays the document.
5. From the **File** menu, choose **Print** and specify the pages that you want to print (and other print options).

Viewing Online Help

Invoke online Help from within VisualDSP++ or outside of VisualDSP++ by clicking the Windows **Start** button. You can also access Help manually via Windows Explorer.

To access online Help from the VisualDSP++ **Help** menu, choose **Contents**, **Search**, or **Index**.

To access online Help from the Windows **Start** button, click the **Start** button and choose **Programs**, **Analog Devices**, **VisualDSP++<version>**, and **VisualDSP++ Documentation**.

The Help function is programmed to look for the Help system in the VisualDSP++ `Help` folder.

By default, the VisualDSP++ software installation procedure places the complete set of Help files in the installation's `Help` folder.

If you receive an error message after invoking Help, the Help system:

- May not have been loaded onto your PC
- May have been deleted
- May reside in a directory other than the default directory

To locate the help (`.CHM`) files manually, use the Windows **Search** function as follows.

1. Record the Help file (`.CHM`) named in the error message.
2. From the Windows **Start** button, choose **Search** and **For Files or Folders**. Enter the name of the `.CHM` file from step 1.
3. After locating the file, launch it manually by clicking the file name from the **Search Results** window (or from Windows Explorer).

Online Help

This section describes online Help features and explains how to:

- Invoke online Help
- View context-sensitive Help
- Copy example code from Help
- Print Help
- Bookmark frequently used help topics
- Navigate in online Help
- Use the search features
- View and print online manuals
- Use the **About VisualDSP++** dialog box

Help Window

The Help window comprises three parts:

- The *Navigation pane* provides tabbed pages (**Contents**, **Index**, **Search**, and **Favorites**) that provide different views.
- The *Viewing pane* displays the selected object (topic, Web page, video, `.PDF` file, application).
- *Toolbar buttons* provide navigation and allow you to specify options.

Figure A-3 shows the parts of the **VisualDSP++ Help** window.

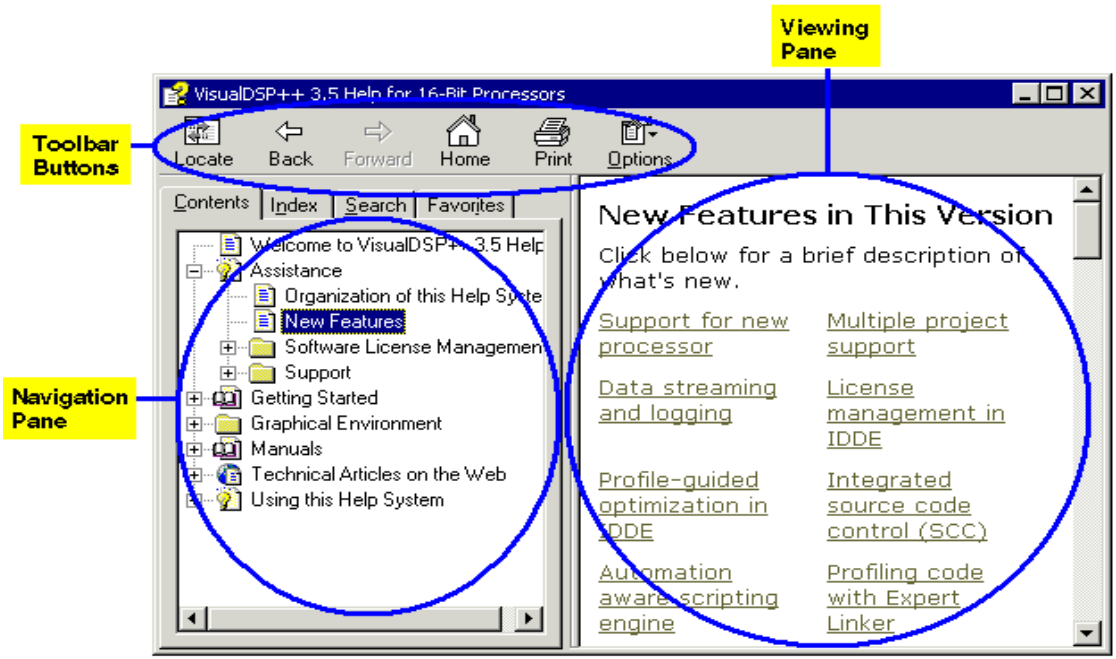

Figure A-3. Parts of the VisualDSP++ Help Window

Move through the Help system and view Help topics by using the Help window's navigational aids, as shown in Figure A-4.

Other standard Microsoft HTML Help buttons are described in Table A-20.

Context-Sensitive Help

You can view context-sensitive Help (information pertinent to your current activity) for various items (toolbar buttons, menu commands, windows, and dialog box items) in VisualDSP++.

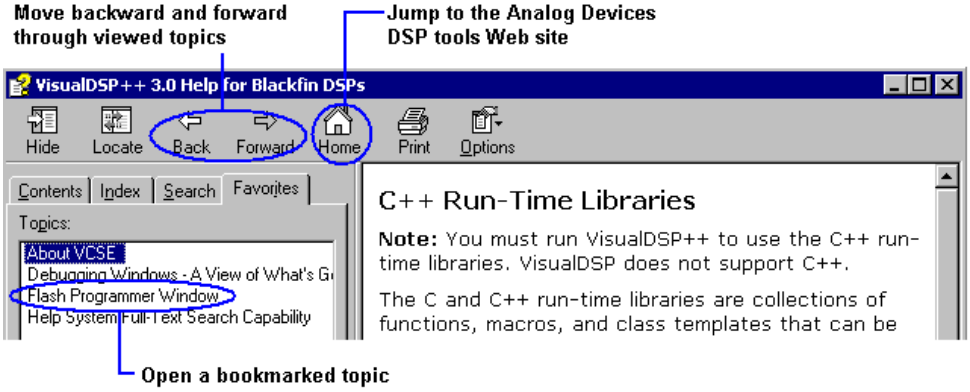
Move backward and forward through viewed topics
Jump to the Analog Devices DSP tools Web site
VisualDSP++ 3.0 Help for Blackfin DSPs
Hide
Locate
Back
Forward
Home
Print
Options
Contents
Index
Search
Favorites
Topics:
About VCSE
Debugging Windows - A View of What's G
Flash Programmer Window
Help System Full-Text Search Capability
C++ Run-Time Libraries
Note: You must run VisualDSP++ to use the C++ run-time libraries. VisualDSP does not support C++.
The C and C++ run-time libraries are collections of functions, macros, and class templates that can be
Open a bookmarked topic

Figure A-4. Help Window Navigational Aids

Table A-20. Standard Microsoft HTML Help Buttons

Button	Purpose
	Hides the Help window's left pane. This button narrows the Help window.
	Displays the Help window's left pane. This button restores a full view after you click **Hide**.
	Highlights the name of the current topic on the **Contents** pane. After you jump around the Help system, this button shows the current topic's relation to other topics. **Note:** The Locate mechanism does not operate at all times.

Viewing Menu, Toolbar, or Window Help

To view menu, toolbar, or window Help:

1. Click the toolbar's Help button ( ) or press **Shift+F1**.

   The mouse pointer becomes a Help pointer ( ).

2. Move the Help pointer over a menu command, toolbar button, or window.

3. Click the mouse to open the Help window. A description of the object appears in the right panel.

Viewing Dialog Box Help

To view Help for a button or field in a dialog box:

- Select a field or button in a dialog box and press **F1** or **Shift+F1**.
- Click the question mark button ( ) in the top-right corner of the dialog box.

  The mouse pointer becomes a Help pointer ( ).

  Next, move the Help pointer over a dialog box control (button or field) and click the mouse. A description of the object appears in a yellow pop-up window.
- Position the mouse pointer over a label or control (button or field) in a dialog box and right-click.

  A **What's This** button ( ) appears. Move the mouse pointer over the **What's This** button and click.

"What's This" Help is not configured for all items.

Viewing Window Help

To view window Help:

1. Click the window to make it active.
2. Press the **F1** key to open the Help window.

   A description of the window appears in the right pane.

Copying Example Code From Help

You can copy code from the Help system and then paste it into your application. Be aware that the copied text may carry unwanted control codes. For example, if you copy a hyphen with a parameter, the actual code of the copied hyphen may be an ASCII `0x96` instead of an ASCII `0x2D`. The hyphen may look OK, but it will cause an error when the command is run.

Printing Help

You can print a specific Help topic or multiple Help topics (an entire section of online Help).

Table A-21. How to Print Help Topics

To print	Do this
Current topic	Right-click within the help topic and choose **Print**.
Selected topic	On the **Contents** page: Right-click the topic ( ) and choose **Print**.
Entire section of Help	On the **Contents** page: Right-click a book icon (or) and choose **Print**. Then choose **Print the selected heading and all subtopics**.

Tip: From the Help window's **Contents** page, click ( ), located at the top of the window.

Bookmarking Frequently Used Help Topics

Bookmarking a topic in online Help is just like bookmarking a page in a book. This feature is also called "setting up favorite places."

Note: Each time a Microsoft HTML Help topic is bookmarked, a record is recorded in the file, `HH.DAT`. This file not only records VisualDSP++ Help bookmarks, but also the bookmarks placed in other application Help systems that use `.CHM` files.

Once a bookmark is placed onto a topic, you can view a list of bookmarked topics and quickly open one.

To place a bookmark at a topic:

1. Display the topic.
2. On the left side of the Help window, click the **Favorites** tab.
3. Click **Add**.

   Remove a bookmark by selecting the name and clicking **Remove**.

   The Help system adds the topic and displays it in the alphabetized list.

To open a bookmarked topic:

1. On the left side of the Help window, click the **Favorites** tab.
2. Perform one of these actions:
   - Double-click the topic.
   - Select the topic and click **Display**.

Navigating in Online Help

To move around in the Help system, click the following.

- **A hyperlink within text.** The text is underlined and displayed in a color that is different from the regular black text.
- **A topic listed under a See Also heading.** The text is underlined and displayed in a color that is different from the regular black text.
- **A mini button or its associated text.** The button is a small gray square and the underlined text is in a different color.
- **A topic name on the Contents page** (Figure A-5)

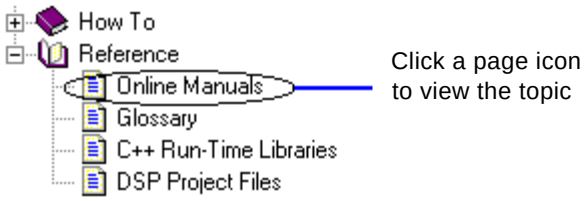

Figure A-5. Contents Page – Online Manual Topics

- **An index entry on the Index page** (Figure A-6)

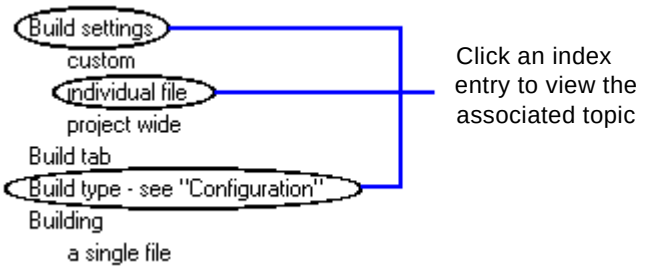

Figure A-6. Index Entries on the Index Page

- **A topic name on the Search page.** The bottom portion of the **Search** page displays the located topics (hits) that include your search string.

Searching Help

VisualDSP++ Help provides full-text and advanced search capabilities for finding information.

Full-Text Searches

A full-text search locates every occurrence of a text string within the Help system. Specify a particular word or phrase to find only the topics that contain that word or phrase.

You can search previous results, match similar words, and search through the topic titles only.

A basic search consists of the word or phrase that you want to locate. Use similar word matches, a previous results list, or topic titles to further define your search.

You can run an advanced search, which uses Boolean operators and wild-card expressions to further narrow the search criteria. Figure A-7 on page A-59 shows an example of a Boolean search for "`new AND plot`".

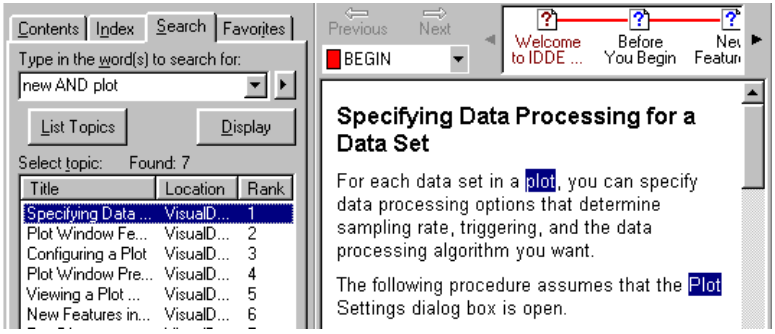
Contents
Index
Search
Favorites
Type in the word(s) to search for:
new AND plot
List Topics
Display
Select topic:
Found: 7
Title
Location
Rank
Specifying Data ... VisualD... 1
Plot Window Fe... VisualD... 2
Configuring a Plot VisualD... 3
Plot Window Pre... VisualD... 4
Viewing a Plot ... VisualD... 5
New Features in... VisualD... 6
Previous
Next
BEGIN
Welcome to IDDE ...
Before You Begin
Specifying Data Processing for a Data Set
For each data set in a plot, you can specify data processing options that determine sampling rate, triggering, and the data processing algorithm you want.
The following procedure assumes that the Plot Settings dialog box is open.

Figure A-7. Boolean Search for "new AND plot"

To find information with a full-text search:

1. Click the Help viewer's **Search** tab.
2. In **Type in the word(s) to search for**, type the word or phrase you want to find.
3. Select **Search previous results** to narrow the search.
4. Select **Match similar words** to find words that are similar to the search string.
5. Select **Search titles only** to search only the topic titles.
6. Click the **Options** button ( ) at the top of the Help Viewer window to highlight all instances of search terms found in topic files. Then choose **Search Highlight On**.
7. Click **List Topics**, select the topic you want, and then click **Display**.

   Note that you can sort the topic list by clicking the **Title**, **Location**, or **Rank** column heading.

Rules for Full-Text Searches

Observe these rules when formulating queries:

- Searches are not case sensitive. Type your search in uppercase or lowercase characters.

  You can search for any combination of letters (a–z) and numbers (0–9).

- Searches ignore punctuation marks such as the period, colon, semicolon, comma, and hyphen.

- Group the elements of your search by using double quotes or parentheses to set apart each element.
- You cannot search for quotation marks.

When searching for a file name with an extension, group the entire string in double quotes (for example, "`filename.ext`"). Otherwise, the period breaks the file name into two separate terms. The default operation between terms is `AND`, which creates the logical equivalent to `filename AND ext`.

Advanced Search Techniques

Use the following search techniques to narrow your searches for more precise results.

- Wildcard expressions
- Boolean operators
- Nested expressions

Wildcard Expressions

Wildcard expressions let you search for one or more characters by using a question mark or asterisk. Table A-22 describes the results of these different kinds of searches.

Table A-22. How to Use Wildcard Expressions to Define a Search

Search Target	Example	Results
A single word	`project`	Locates topics that contain the word "project". Other grammatical variations, such as "projects" are located.

Table A-22. How to Use Wildcard Expressions to Define a Search

A phrase	"project window" (note the quotation characters) project window	Locates topics that contain the literal phrase "project window" and all its grammatical variations. Without the quotation characters, the query is equivalent to specifying "project AND window", which finds topics containing both of the individual words, instead of the phrase.
Wildcard expressions	link* -or- .C??	Locates topics that contain the terms "linker", "linking", "links", and so on. The asterisk cannot be the only character in the term. Locates topics that contain the terms ".CPP" or ".CXX". The question mark cannot be the only character in the term.

Boolean Operators

Use the Boolean AND, OR, NOT, and NEAR operators to precisely define your search by creating a relationship between search terms.

Insert a Boolean operator by typing the operator (AND, OR, NOT, or NEAR) or by clicking the arrow button.

When no operator is specified, AND is used. For example, the query `call stack` is equivalent to `call AND stack`.

Table A-23 describes the results of using Boolean operators to define a search.

Table A-23. Examples: Boolean Operators Used to Define a Search

Search Target	Example	Results
Both terms in the same topic	`new AND plot`	Locates topics that contain both the words “new” and “plot”
Either term in a topic	`new OR plot`	Locates topics that contain either the word “new” or the word “plot” or both
The first term without the second term	`new NOT plot`	Locates topics that contain the word “new”, but not the word “plot”
Both terms in the same topic, close together	`new NEAR plot`	Locates topics that contain the word “new” within eight words of the word “plot”

Do not use the |, &, or ! characters as Boolean operators. You must use OR, AND, or NOT.

Nested Expressions

Use nested expressions to create complex searches for information. For example, `new AND ((plot OR waterfall) NEAR window)` finds topics containing the word “new” along with the words “plot” and “window” close together, or topics containing “new” along with the words “waterfall” and “window” close together.

Rules for Advanced Searches

These rules apply to advanced searches:

- Expressions in parentheses are evaluated before the rest of the query.
- If a query does not contain a nested expression, it is evaluated from left to right. For example, “`folder NOT file OR project`” finds topics containing the word “folder” without the word “file,” or topics containing the word “project”. The expression “`folder NOT (file OR project)`”, however, finds topics containing the word “folder” without either of the words “file” or “project.”
- You cannot nest expressions deeper than five levels.

About VisualDSP++ Dialog Box

Choosing the **About VisualDSP++** command in the **Help** menu opens the **About VisualDSP++** dialog box, which provides access to the following types of support information.

- Software versions

  The **General** page (Figure A-8) provides information about your version of VisualDSP++. This information includes the name of the registered user and company, the version of IDDE and its build date, and the directory path in which VisualDSP++ is installed.

Figure A-8. Example of the General Page

- License management

  The **Licenses** page (Figure A-9) provides a centralized view of your current licenses. You can view license status and perform all necessary licensing activities (installing, registering, and validating).

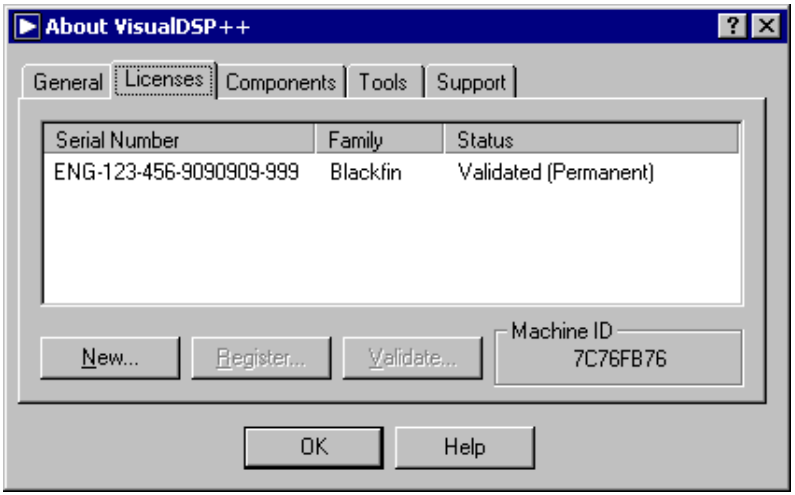

Figure A-9. Example of the Licenses Page

- Component versions

  The Components page (Figure A-10) displays a list of your system's components and provides information (name, version, provider) about your debug target, symbol manager, and processor library.

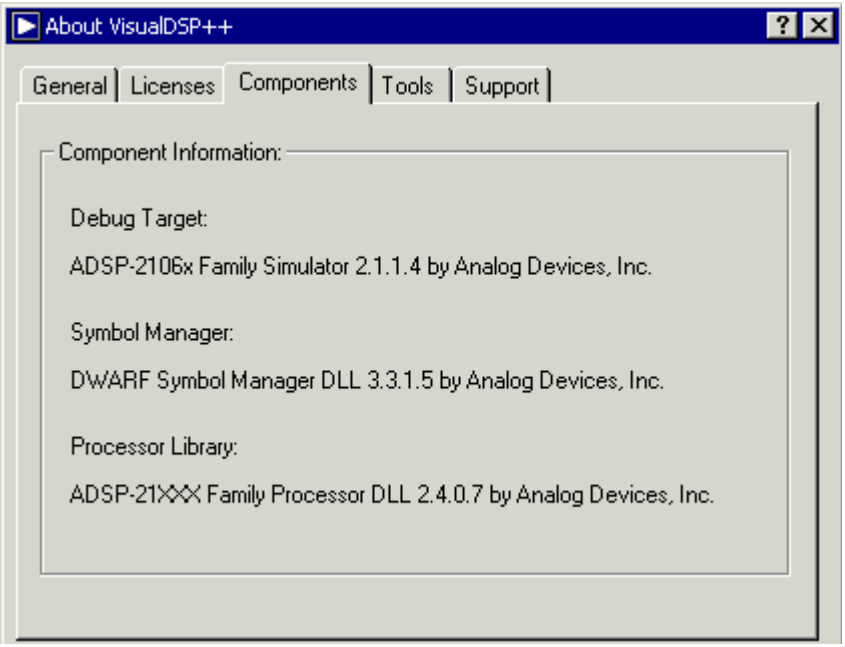

Figure A-10. Example of the Components Page

- Tools

  The **Tools** page (Figure A-11) displays a list of your system's tools. Each tool includes a description, version number, and the name of the company that developed it.

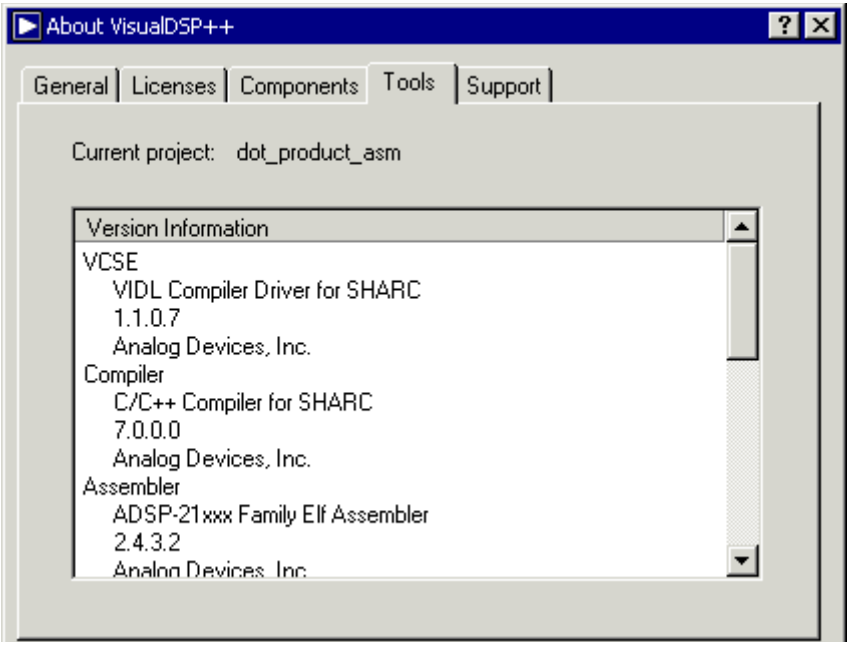

Figure A-11. Example of the Tools Page

- Links to support on the Web

  The **Support** page (Figure A-12) provides direct links to various Web pages that contain support information such as application notes, code examples, the DSP Knowledgebase, processor and tools anomalies and workarounds, manuals and data sheets, product comparisons, tools updates, and more. You can also generate the body of an e-mail that automatically contains your system's description.

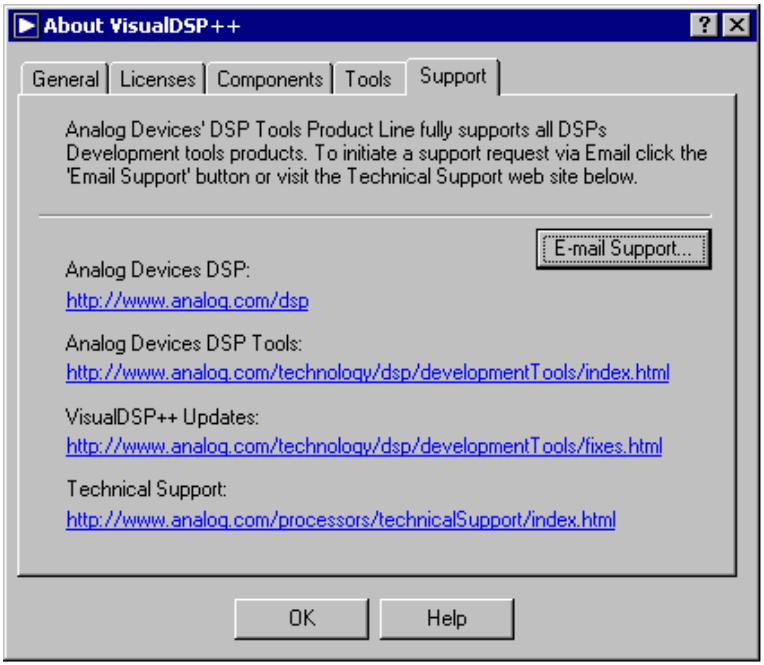

Figure A-12. Example of the Support Page

Window Operations

Similar to many Windows applications, VisualDSP++ provides multiple ways to adjust the view of the user interface.

Window Manipulation

The **Window** menu commands (Figure A-13) enable you to manipulate your window display and update windows during program execution. Refer to your Windows documentation for more information.

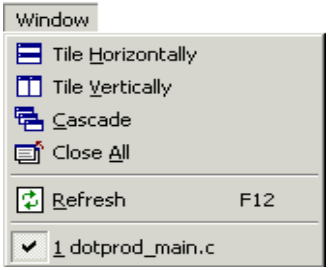

Figure A-13. Window Menu Commands

Right-Click Menu Options

A menu appears when you right-click in a window or on its title bar. The menu options in Table A-24 affect window behavior.

Table A-24. Window Right-Click Menu Commands

Option	Description
Allow Docking	Enables or disables docking
Close	Closes the window
Float in Main Window	Causes the window to become a normal MDI child window (like an editor window) and disables its docking ability

Scroll Bars and Resize Pull-Tab

Scroll bars appear along the right and bottom edges of the application or document window, as shown in Figure A-14.

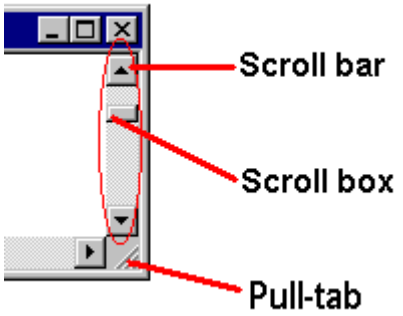

Figure A-14. Scrolling the View Area

The scroll boxes inside the scroll bars indicate the vertical and horizontal location in the document. Use the mouse to scroll to other parts of the document.

When the application window is not maximized, the resize pull-tab appears in the lower-right corner of the window. Click and drag the pull-tab to resize the application window.

Windows: Docked vs. Floating

A window attached to the application's frame is referred to as a *docked window*.

You can detach a window from the main window and move it to another location anywhere on the desktop. A *floating window* stands alone, because it is not docked.

Depending on your needs, you can:

- Dock a window to the application's main window (frame)
- Float a window

A window's right-click menu provides commands for docking or floating the window. The **Allow Docking** command and the **Float In Main Window** commands are mutually exclusive.

Docked Windows

The **Project** window in Figure A-15 is docked. (The docking option, **Allow Docking**, is enabled.)

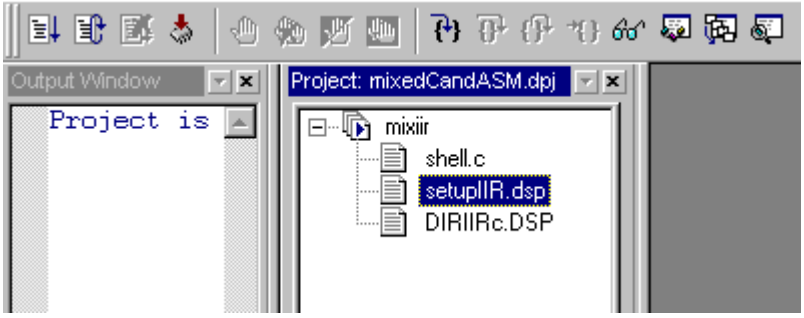

Figure A-15. Example of a Docked Project Window

To prevent a window from docking, hold down the keyboard's **Ctrl** key while dragging the window to another position.

Floating Windows

The **Project** window in Figure A-16 is floating in the main window. (**Float In Main Window** is enabled.)

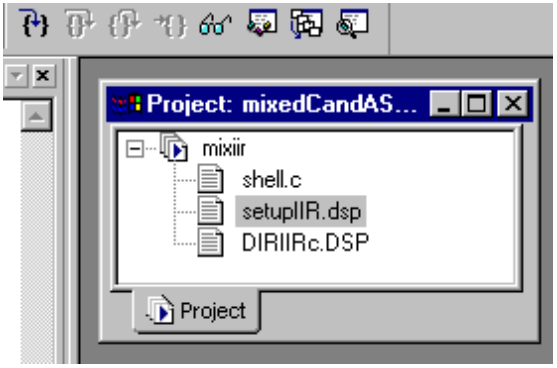

Figure A-16. Project Window Floating in Main Window (1 of 2)

The **Project** window in Figure A-17 is also floating in the main window. (**Float In Main Window** is enabled.)

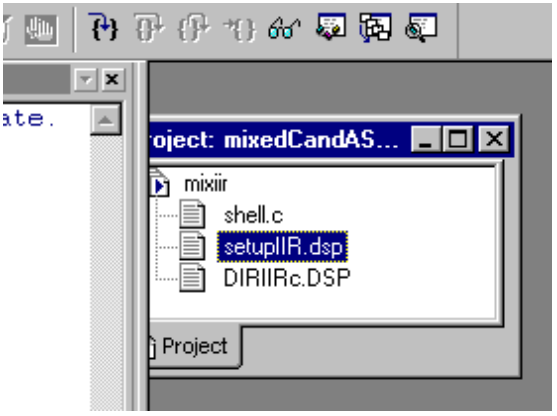

Figure A-17. Project Window Floating in Main Window (2 of 2)

The Project window in Figure A-18 is floating, but not in the main window. (**Float In Main Window** is not selected.)

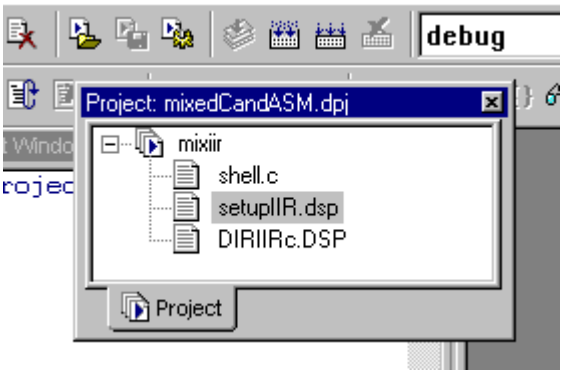

Figure A-18. Example: Project Window is Not Floating in Main Window

Window Position Rules

The following rules apply to window positions.

- Unless **Allow Docking** is disabled, a window must reside within the main window.
- An editor window cannot be docked to the main window.
- A window specified as an MDI child cannot be positioned over a docked window.
- Unless the **Output** window is floating in the main window, a window specified as an MDI child cannot be positioned over the **Output** window.

Standard Windows Buttons

The standard Windows buttons are located on the right side of the title bar, as shown in Figure A-19.

Figure A-19. Example: Title Bar Showing Standard Window Buttons

These buttons resize and close the window as described in Table A-25.

Table A-25. Standard Windows Buttons

Button	Name – Purpose
	Minimize – reduces the window to its Windows icon
	Maximize – enlarges the window to fill the screen
	Restore – returns the window to its last non-minimized, non-maximized position after you maximize the window
	Close – closes the application window and exits the program

B SIMULATION OF SHARC PROCESSORS

Depending on your selected target processor, several simulator options are available on submenus under the **Settings** menu.

This appendix describes the options that help you simulate SHARC processors.

The information is organized as follows.

- "Anomaly Options" on page B-2
- "Event Options" on page B-4
- "Recording a Simulator Anomaly or Event" on page B-7
- "Select Processor ID Options" on page B-10
- "Simulator Options" on page B-10
- "Load Sim Loader Options" on page B-11
- "SPI Simulation in Slave Mode" on page B-13

Anomaly Options

The **Anomalies** submenu (under **Settings** menu) provides commands to help you determine where an anomaly might be affecting your code. This submenu appears only when the simulator supports anomaly commands for the selected processor.

ADSP-21x6x Processor Anomalies

If you know that your silicon has anomalies, the simulator lets you configure reporting for the following anomaly events. By default, these options are OFF (disabled).

- **Shadow Write** – This command opens the **Configure Simulator Event** dialog box, from which to configure reporting for Shadow Write anomalies.
- **SIMD FIFO** – This command opens the **Configure Simulator Event** dialog box, from which to configure reporting for SIMD FIFO anomalies.

Shadow Write FIFO Anomaly (ADSP-2116x Only)

For examples and workarounds, refer to anomaly 39 at the Analog Devices embedded processors and DSPs Web site.

This anomaly has been identified in the shadow write FIFOs that exist between the internal memory array of the ADSP-21160M and core I/O processor (IOP) buses that access the memory. (Refer to the hardware documentation for more details on shadow register operation.) A particular sequence of a core write followed by a read of the same internal memory address, in conjunction with a certain type of IOP activity can cause the core read to return incorrect data.

Under the circumstances described below, the Read from `Addr 1` incorrectly returns the data for `Addr 2`.

This problem is caused by the shadow write FIFO erroneously returning data for a core read when data should have been returned from internal memory. During write operations, data is placed in the first stage of a two-stage shadow write FIFO. Data is moved from first to second stage when a second write is performed (by either processor core or IOP). Similarly, data is moved from the second stage of the FIFO to internal memory when neither the core nor the IOP accesses memory in a core cycle.

On read operations, address compare logic allows data to be fetched from internal memory or from the FIFOs. Note that each memory block has one Shadow Register FIFO, and all core and IOP accesses to internal memory use this FIFO. The internal memory clock (not visible to the user) runs at twice the core clock frequency. So, each core cycle consists of two memory cycles, with one memory cycle dedicated to the core and the other dedicated to the IOP.

SIMD Read from Internal Memory With Shadow Write FIFO Hit Anomaly (ADSP-2116x Only)

For examples and workarounds, refer to anomaly 40 at the Analog Devices Web site.

This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M and core /IOP buses that access the memory. (Refer to the hardware documentation for details on shadow register operation.)

When SIMD reads cross Long Word Address boundaries (that is, odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the result is Revision 0.0 behavior for the read.

Event Options

The **Events** submenu provides the following options.

- **FP Denorm** – This command opens the **Configure Simulator Event** dialog box, from which to configure the reporting of the generation of a floating-point denormal result. By default, this option is OFF (disabled).
- **Short Word Anomaly** (all ADSP-2106*x* processors except the ADSP-21065L processor) – This command opens the **Configure Simulator Event** dialog box, from which to configure reporting for short-word accesses that fail. By default, this option is OFF (disabled).
- **Access to 21065L 9th column Even Address** (ADSP-21065L processors only) – This command opens the **Configure Simulator Event** dialog box, from which to configure reporting for invalid memory access. By default, this option is ON (enabled).

FP Denorm

You can configure what happens when an **FP Denorm** event occurs. Denormal operands flush to zero when input to a computation unit and do not generate an underflow exception. Refer to your processor's hardware documentation for more information about floating-point operations.

Short Word Anomaly

This option applies to all ADSP-2106x processors except the ADSP-21065L processor.

Short-word accesses (read or write) fail following a stalled instruction. Any access (read or write) to short-word memory space can fail if it follows a stalled instruction or causes an instruction stall (see below for examples).

A DMA process cannot cause this anomaly. It is restricted to conditions set up in the core processor. A DMA process is not impacted by this anomaly (that is, the DMA will function correctly even if the anomaly occurs). Refer to your processor's hardware documentation.

The occurrence of the failure varies with temperature, voltage, and frequency.

An instruction can stall because of the following circumstances.

1. DAG Stalls

   An instruction that loads a DAG register followed by an instruction that uses the same DAG for a memory access causes the second instruction to stall.

   ```
   L2= 8;
   DM(I0, M0) = R1;      //  Both L2 and I0 reside in DAG1,
                         //  causing this instruction
                         //  to stall.
   ```

   Failure can occur if `I0` points to a short-word space or if the above instruction sequence is followed by a short word access.

2. PM Memory Data Access (Cache Misses)

   Any instruction that uses the PM bus to perform a data access causes an instruction stall the first time it is executed.

   ```
   PM(I8,M8) = R1;
   ```

3. Memory Block Conflicts

   If an instruction requires two accesses to the same memory block, the instruction stalls.

   ```
   DM(I0, M0)=R0, PM (I8,M8) = R1;
       //  A stall will occur when both address
       //  pointers point to the same memory block.
   ```

4. Wait States for External Memory Accesses

   An instruction that contains an external memory access where the wait state setting for that memory bank is greater than 0 or the access is held off because of the ACK signal or through bus arbitration causes that instruction to stall.

5. Multiple Bus Accesses to IOP Registers in the Same IOP Register Group

   An instruction may be stalled because of multiple buses trying to access IOP registers in the same group. For this anomaly, the only applicable case is if an external host or processor accesses the same group of registers at the same time as the core.

```
DM(MSGR0) = R1;     //  Instruction executed while
                    //  the host or another processor
                    //  writes to MSGR2.
```

6. Executing Instructions from External Memory with Wait States

   Any instruction being executed from external memory where the wait state setting for that memory bank is greater than 0 or the access is held off because of the `ACK` signal causes that instruction to stall.

A workaround for cases 1–5 above is to insert a `NOP` between the stallable instruction and the short-word access (or remove the stall condition).

Case 6 has no workaround.

Ensure that a failing sequence does not occur when you use the delayed branch (`DB`) option with jumps, calls, and returns. For example, ensure that the two instructions in the `RTI (DB)` do not cause an instruction stall in a return to code that includes short-word accesses.

Access to ADSP-21065L Short-Word Internal Memory 9th Column at Even Addresses

An **Access to ADSP-21065L 9th column Even Address** event is a simulator anomaly. The event occurs during access to an ADSP-21065L short-word internal memory 9th column even addresses. The simulator allows the event, but the processor does not.

The simulator issues a warning when the event occurs. VisualDSP++ lets you suppress the warning (in various ways). Selecting this option lets you control the simulator's behavior when the event occurs. When this option is not selected, a message box pops up and a warning appears in the **Output** window's **Console** view.

Recording a Simulator Anomaly or Event

You can record various simulator anomalies and events for ADSP-21x6x processors.

To record a simulator anomaly or event:

1. From the **Settings** menu, choose **Anomalies** or **Events**.
2. Choose the anomaly or event to record.

   The anomalies are: **Shadow Write** or **SIMD FIFO**

   The events are: **FP Denorm**, **Short Word Anomaly**, or **Access to 21065L 9th column Even Address**

   The **Configure Simulator Event** dialog box (Figure B-1) appears, displaying the selected anomaly or event (for example, **FP Denorm**) in the title bar of the dialog box.

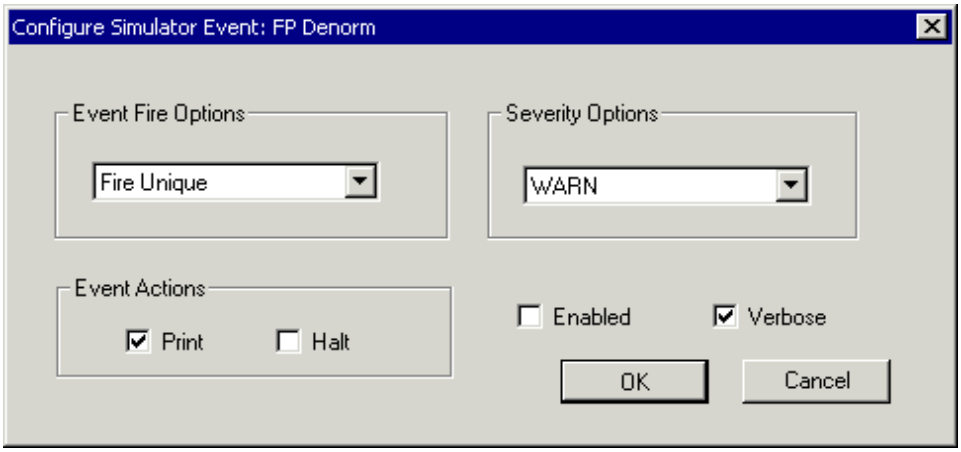

Figure B-1. Configure Simulator Event Dialog Box

From this dialog box, configure the simulator to handle the anomaly or event.

3. Specify options, described in Table B-1.

Table B-1. Options in the Configure Simulator Event Dialog Box

Item	Purpose
Event Fire Options	These options specify the frequency of reporting an event. **Fire Once** logs the event only the first time it occurs. **Fire Unique** logs the event once for each unique event (a unique set of occurrences which is specific to the event type). Select this option to prevent the reporting of multiple messages for the same event. Events (for example, **FP Denorm**) – If a PC has had this event before, it is not unique and is not reported. Anomalies (for example, **Shadow Write**, **SIMD FIFO**, or **Short Word**) – If the PC at the time of the memory write and the PC at the time of the memory read taken as a pair have had this event before, it is not unique and is not reported. **Fire All** logs every occurrence of the event.
Severity	This option specifies the degree of the event. **INFO** writes a message in black typeface to the **Output** window. **WARN** writes a message in black typeface to the **Output** window. **ERROR** writes a message in red typeface to the **Output** window and rings a bell. **FATAL** writes a message in red typeface to the **Output** window and rings a bell.
Event Actions	**Print** writes messages to the **Output** window. **Halt** stops processing after the event has occurred. This option is similar to using a watchpoint.
Enabled	Enables this event check
Verbose	Specifies that multi-line messages are written to the **Output** window When this option is not selected, messages are one line long.

4. Click **OK**.

Select Processor ID Options

For ADSP-2106x or ADSP-2116x processors only, you can configure the simulator processor ID. Select **Single Processor** or a particular processor in a multiprocessor group (for example: **Processor 1**, **Processor 2**, and so on).

Simulator Options

The **Simulator** submenu (under the **Settings** menu) provides the **CLKDBL** command for ADSP-21161 processors only. Use this command to double the clock speed circuitry.

Clock doubling lets you set control bits, but it does not affect how the simulator runs.

You can configure the processor's `CLKOUT` pin to be 1*x* or 2*x* the rate of `CLKIN`. The appearance of a check mark (✓) beside the **CLKDBL** command on the **Simulator** submenu indicates that this option is selected.

No Boot Mode

(**ADSP-2106x and ADSP-2116x processors only**).
The **Simulation** submenu's **NoBootMode** command is available for ADSP-21x6x simulators to support development code in configurations where the processor is in "NoBoot" mode (starting execution from external memory for pre-ADSP-2126x processors, or starting execution from internal ROM for ADSP-2126x and ADSP-2136x processors).

When this mode is selected, a check mark appears beside the **NoBootMode** command in the **Simulation** submenu. Also, a message displays, reminding you to load a program or choose **Debug** -> **Reset** to complete the mode change.

Changing to NoBoot Mode is not persistent. Each time you bring up the simulator and/or change sessions, the mode reverts back to its default state, which is Boot Mode.

The processor has a “boot” and a “NoBoot” mode. In boot mode, the IIVT bit is ignored and the interrupt vector table (IVT) and starting PC are in internal memory. In NoBoot mode, the IIVT bit has significance, and the initial location of the IVT and PC are in external memory (pre ADSP-2126x processors) and in internal ROM (ADSP-2126x processors and later).

Before VisualDSP++ 4.0, all of the SHARC (ADSP-21x6x) simulators supported “Boot” mode only, regardless whether an `.LDF` file was being booted because most users develop code whose final form is booted into the processor.

Load Sim Loader Options

Depending on the target processor, the **Load Sim Loader** submenu provides these boot options:

- Boot from Host (32-bit Host, 16-bit Host, or 8-bit Host)
- Boot from PROM
- Boot from Link (ADSP-21060, ADSP-21062, and ADSP-2116x processors only)
- Boot from SPI (32-bit Host, 16-bit Host, or 8-bit Host) for ADSP-2116x processors only
- None of Above (disables boot mode)

Create a boot loader file (.LDR) based on the peripheral from which you are loading. In a simulation target session, choose a peripheral (boot option) and boot file as follows.

1. From the **Settings** menu, choose **Load Sim Loader** and a boot option (such as **Boot from Host**) to open the **Open a Boot File** dialog box.
2. Navigate to the .LDR file to boot load and select it. Then click **Open** and **OK**. A message instructs you to issue a reset instruction to execute the loader.
3. From the **Debug** menu, choose **Reset**. The simulator runs and boots in the boot-kernel (also called loader-kernel). A message instructs you to issue a run instruction, which executes the loader.
4. From the **Debug** menu, choose **Run** (**F5**) to execute the loader. The loader runs to completion and then displays a message, indicating that the loader is finished and that it is OK to run the application.

Before clicking **Run**, load the symbols for the program as follows.

   a. From the **File** menu, choose **Load Symbols** to open the **Load a Processor Program's Symbols** dialog box.

   b. Select the .DXE file that was used to create the .LDR file. A message indicates that the selected symbols are loading.

5. From the **Debug** menu, choose **Run** (**F5**) to run the application.

For booting information, refer to the processor's hardware documentation or the *VisualDSP++ 4.0 Linker and Utilities Manual*.

SPI Simulation in Slave Mode

For ADSP-21161 processors, the external SPI is not modeled in simulation. Since the master controls the timing, there is no timing information for the transfers in slave mode. Except during booting, the BAUDR field of the slave SPICTL is used for the timing of the external master. Since this field has no effect in slave mode in the hardware, you can use it for simulation.

When the SPI is enabled in slave mode and the SPITX buffer is not empty, an SPI transmit/receive operation occurs. If the SPITX and the SPIRX buffers are empty, a word is read speculatively into SPIRX buffer. The SPI reads a maximum of one word ahead. Once the SPIRX buffer is emptied by a DMA operation or a read, the SPI reads another word in the time specified by the BAUDR field of SPICTL.

During booting, the SPI operates at its fastest supported baud rate. When you rely on the BAUDR field in slave mode, a maximum value of 0x5 is used for this field. The “End of file” and “File not connected” errors are suppressed until a read of SPIRX or a valid DMA is set up.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for details.

C SIMULATION OF TIGERSHARC PROCESSORS

This appendix describes how to simulate TigerSHARC processors.

The information is organized as follows.

- "ADSP-TS101 Processors" on page C-1
- "ADSP-TS20x Processors" on page C-13

ADSP-TS101 Processors

This section includes the following topics, which apply to ADSP-TS101 processors.

- "Simulator Timing Analysis Overview" on page C-2
- "Pipeline Stages" on page C-2
- "Stalls" on page C-3
- "Aborts" on page C-6
- "Pipeline Viewer and Disassembly Window Operations" on page C-8
- "Simulator Options" on page C-12

Simulator Timing Analysis Overview

The ADSP-TS101 simulator is a cycle-accurate simulator. It not only models the instruction set functionally, but also correctly models pipeline effects (stalls and aborts).

Currently, the processor's external port and link ports are not modeled in a cycle-accurate manner. The simulator cycle-counts the code, but the cycle counts used for the external port or link ports are rough estimates of cycle counts that you could obtain by running the code on the chip. Do not rely on these counts for performance evaluation.

The **Pipeline Viewer** window helps you understand how processor timing affects the execution of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-94 or VisualDSP++ online Help.

Pipeline Stages

The ADSP-TS101 **Pipeline Viewer** window provides a representation of instruction flow through the processor's pipeline.Table C-1 lists the pipeline stages.

Table C-1. Pipeline Stages – ADSP-TS101 Processor

Stage	Abbreviation in Pipeline Viewer
Instruction Fetch 1	F1
Instruction Fetch 2	F2
Instruction Fetch 3	F3
Decode	DECODE
Integer	INT

Table C-1. Pipeline Stages – ADSP-TS101 Processor (Cont'd)

Stage	Abbreviation in Pipeline Viewer
Access	ACCESS
Execute Stage 1	EX1
Execute Stage 2	EX2

Stages `F1` through `F3` represent the fetch pipe. Stages `DECODE` through `EX2` represent the execution pipe. Fetch pipe stages contain raw quads of 32-bit words fetched from memory, not valid instruction lines.

When you view the **Pipeline Viewer** window in disassembly format (by using the right-click menu), instructions that appear in the fetch pipe are not valid instruction lines, but merely bits and pieces. Valid instruction lines appear in the execution pipe stages.

Stalls

The examples that follow illustrate the way Pipeline Viewer displays different types of stall events for ADSP-TS101 processors. For a complete list of pipeline effects and memory transaction timing, refer to the processor's hardware specification.

Stalls Due to IALU Dependency

The following sequence of instructions causes a 4-cycle stall at the Decode stage.

```
J10=0x0;;
xr5 = [j10 + 1];;    //  This instruction is stalled in Decode
                     //  until previous instruction reaches E2:
```

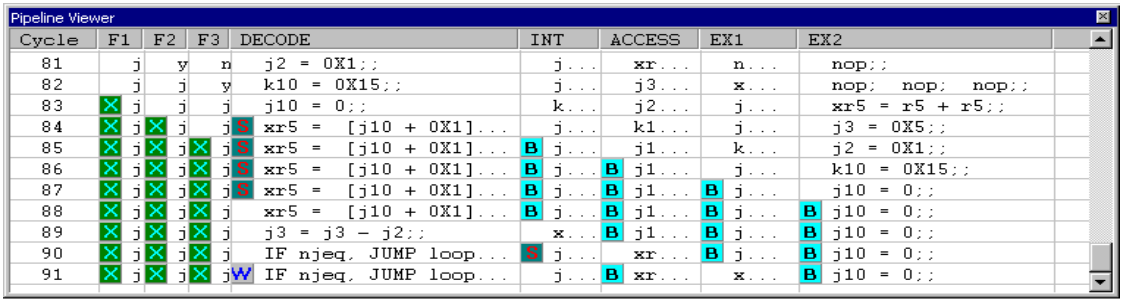
Pipeline Viewer
Cycle F1 F2 F3 DECODE INT ACCESS EX1 EX2
81 j y n j2 = 0X1;; j... xr... n... nop;;
82 j j y k10 = 0X15;; j... j3... x... nop; nop; nop;;
83 X j j j j10 = 0;; k... j2... j... xr5 = r5 + r5;;
84 X j X j j S xr5 = [j10 + 0X1]... j... k1... j... j3 = 0X5;;
85 X j X j X j S xr5 = [j10 + 0X1]... B j... j1... k... j2 = 0X1;;
86 X j X j X j S xr5 = [j10 + 0X1]... B j... B j1... j... k10 = 0X15;;
87 X j X j X j S xr5 = [j10 + 0X1]... B j... B j1... B j... j10 = 0;;
88 X j X j X j xr5 = [j10 + 0X1]... B j... B j1... B j... B j10 = 0;;
89 X j X j X j j3 = j3 - j2;; x... B j1... B j... B j10 = 0;;
90 X j X j X j IF njeq, JUMP loop... S j... xr... B j... B j10 = 0;;
91 X j X j W j IF njeq, JUMP loop... j... B xr... x... B j10 = 0;;

Figure C-1. Stall Due to IALU Dependency

Stalls Due to Compute Block Dependency

The following sequence of instructions causes a one-cycle stall at the Integer stage.

```
yr3 = r3 - r2;;
yr4 = r4 - r3;;     //  This instruction is stalled in Integer for
                    //  1 cycle. The source register (R2) is a
                    //  destination in the previous instruction.
```

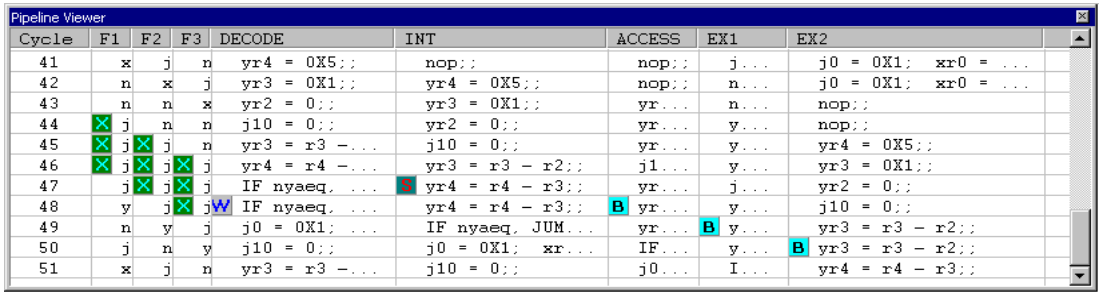

Figure C-2. Stall Due to Compute Block Dependency

Aborts

The following examples show how the Pipeline Viewer displays different types of abort events for ADSP-TS101 processors. For a complete list of pipeline effects and memory transaction timing, refer to the processor's hardware specification.

Aborts Due to an Unpredicted Change of Flow

In the following example, the abort at the Decode stage is due to an unpredicted jump, predicated upon an IALU condition. Aborted stages in the fetch pipe are marked with X, and aborted instructions in the execution pipe are marked with A .

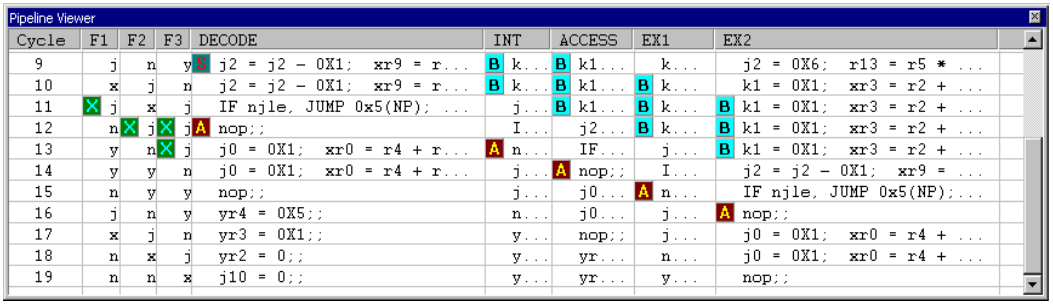
Pipeline Viewer
Cycle F1 F2 F3 DECODE INT ACCESS EX1 EX2
9 j n y j2 = j2 – 0X1; xr9 = r... k... j2 = 0X6; r13 = r5 * ...
10 x j n j2 = j2 – 0X1; xr9 = r... k... k1... k... k1 = 0X1; xr3 = r2 + ...
11 j x j IF njle, JUMP 0x5(NP); ... j... k1... k... k1 = 0X1; xr3 = r2 + ...
12 n j j nop;; I... j2... k... k1 = 0X1; xr3 = r2 + ...
13 y n j j0 = 0X1; xr0 = r4 + r... n... IF... j... k1 = 0X1; xr3 = r2 + ...
14 y y n j0 = 0X1; xr0 = r4 + r... j... nop;; I... j2 = j2 – 0X1; xr9 = ...
15 n y y nop;; j... j0... n... IF njle, JUMP 0x5(NP);...
16 j n y yr4 = 0X5;; n... j0... j... nop;;
17 x j n yr3 = 0X1;; y... nop;; j... j0 = 0X1; xr0 = r4 + ...
18 n x j yr2 = 0;; y... yr... n... j0 = 0X1; xr0 = r4 + ...
19 n n x j10 = 0;; y... yr... y... nop;;

Figure C-3. Abort Due to an Unpredicted Change of Flow

Abort Due to Mispredicted Change of Flow

In the following example, an abort appears at stage E1 because of a mispredicted change of flow at the end of a loop, predicated on a compute block condition.

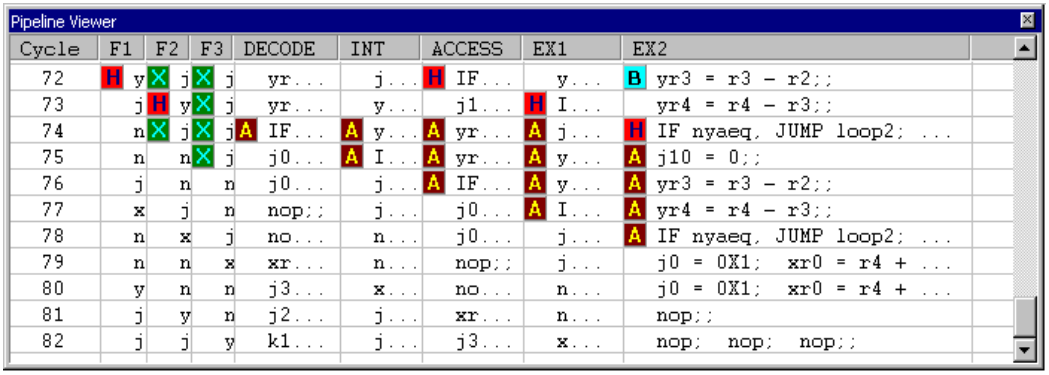
Pipeline Viewer
Cycle F1 F2 F3 DECODE INT ACCESS EX1 EX2
72 H y X j X j yr... j... H IF... y... B yr3 = r3 - r2;;
73 j H y X j yr... y... j1... H I... yr4 = r4 - r3;;
74 n X j X j A IF... A y... A yr... A j... H IF nyaeq, JUMP loop2; ...
75 n n X j j0... A I... A yr... A y... A j10 = 0;;
76 j n n j0... j... A IF... A y... A yr3 = r3 - r2;;
77 x j n nop;; j... j0... A I... A yr4 = r4 - r3;;
78 n x j no... n... j0... j... A IF nyaeq, JUMP loop2; ...
79 n n x xr... n... nop;; j... j0 = 0X1; xr0 = r4 + ...
80 y n n j3... x... no... n... j0 = 0X1; xr0 = r4 + ...
81 j y n j2... j... xr... n... nop;;
82 j j y k1... j... j3... x... nop; nop; nop;;

Figure C-4. Abort Due to Mispredicted Change of Flow

Branch Target Buffer Hits

In the following example, the jump instruction was found in the Branch Target Buffer and is marked with H. The branch is predicted and taken, and no penalty is exacted for the change of flow.

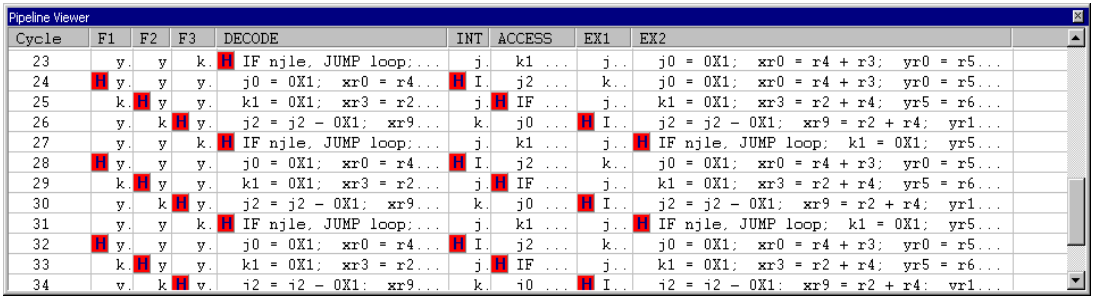
Pipeline Viewer
Cycle F1 F2 F3 DECODE INT ACCESS EX1 EX2
23 y. y k. H IF njle, JUMP loop;... j. k1 ... j.. j0 = 0X1; xr0 = r4 + r3; yr0 = r5...
24 H y. y y. j0 = 0X1; xr0 = r4... H I. j2 ... k.. j0 = 0X1; xr0 = r4 + r3; yr0 = r5...
25 k. H y y. k1 = 0X1; xr3 = r2... j. H IF ... j.. k1 = 0X1; xr3 = r2 + r4; yr5 = r6...
26 y. k H y. j2 = j2 - 0X1; xr9... k. j0 ... H I.. j2 = j2 - 0X1; xr9 = r2 + r4; yr1...
27 y. y k. H IF njle, JUMP loop;... j. k1 ... j.. H IF njle, JUMP loop; k1 = 0X1; yr5...
28 H y. y y. j0 = 0X1; xr0 = r4... H I. j2 ... k.. j0 = 0X1; xr0 = r4 + r3; yr0 = r5...
29 k. H y y. k1 = 0X1; xr3 = r2... j. H IF ... j.. k1 = 0X1; xr3 = r2 + r4; yr5 = r6...
30 y. k H y. j2 = j2 - 0X1; xr9... k. j0 ... H I.. j2 = j2 - 0X1; xr9 = r2 + r4; yr1...
31 y. y k. H IF njle, JUMP loop;... j. k1 ... j.. H IF njle, JUMP loop; k1 = 0X1; yr5...
32 H y. y y. j0 = 0X1; xr0 = r4... H I. j2 ... k.. j0 = 0X1; xr0 = r4 + r3; yr0 = r5...
33 k. H y y. k1 = 0X1; xr3 = r2... j. H IF ... j.. k1 = 0X1; xr3 = r2 + r4; yr5 = r6...
34 y. k H y. j2 = j2 - 0X1; xr9... k. j0 ... H I.. j2 = j2 - 0X1; xr9 = r2 + r4; yr1...

Figure C-5. Branch Target Buffer Hits

Pipeline Viewer and Disassembly Window Operations

This section includes the following topics.

- Current Program Counter Value
- Stepping

Current Program Counter Value

The program counter value, marked by ➪ in the **Disassembly** window, is the address of the instruction at stage `E1`. When a breakpoint is set at a certain address, the simulator halts after the instruction at this address is executed at stage `E1`.

You can specify which instruction the simulator executes next by manually changing the program counter (PC) value in the **PC Register** window. Note, however, that the current instruction, indicated by ➪ in the **Disassembly** window, is executed prior to the user-specified instruction. When you change a PC value, the simulator automatically flushes the pipeline and aborts all its instructions. After the pipeline is flushed, normal execution resumes from a memory address indicated by the new PC value. The simulator continues to run until an instruction at the new PC reaches stage `E1` of the pipeline.

Manually changing the PC value during simulation can result in unpredictable program behavior. VisualDSP++ does not safeguard against invalid PC values. Ensure that the specified PC indicates a valid instruction within program address space.

An example of an invalid PC is a data memory address whose contents is not recognized as an instruction by the simulator. In this case, the simulator generates an unhandled software exception.

Another example of an erroneous PC is the middle of an instruction line. Only part of the instruction line is executed, while instructions in the beginning of the line are ignored.

Figure C-6 shows how the program counter value is used for the ADSP-TS101 processor.

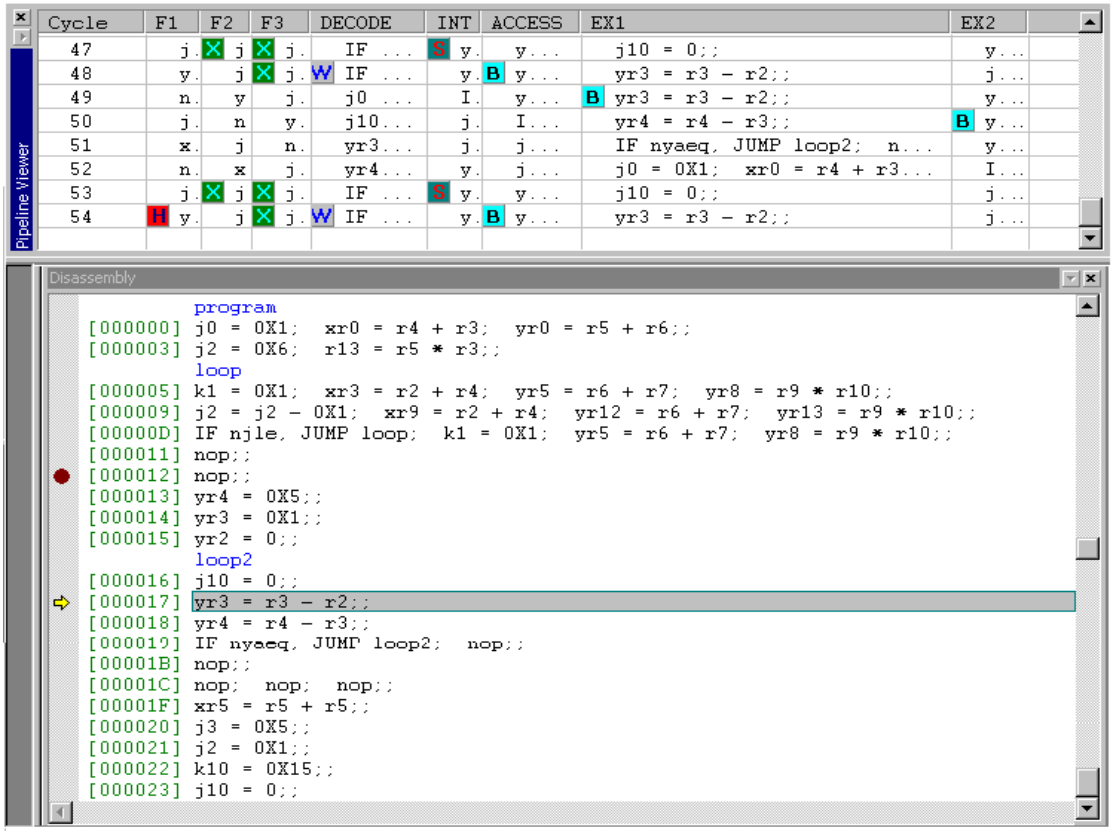
Pipeline Viewer
Cycle F1 F2 F3 DECODE INT ACCESS EX1 EX2
47 IF ... y... j10 = 0;; y...
48 IF ... y... yr3 = r3 - r2;; j...
49 j0 ... I... yr3 = r3 - r2;; y...
50 j10... I... yr4 = r4 - r3;; y...
51 yr3... j... IF nyaeq, JUMP loop2; n... y...
52 yr4... j... j0 = 0X1; xr0 = r4 + r3... I...
53 IF ... y... j10 = 0;; j...
54 IF ... y... yr3 = r3 - r2;; j...
Disassembly
program
[000000] j0 = 0X1; xr0 = r4 + r3; yr0 = r5 + r6;;
[000003] j2 = 0X6; r13 = r5 * r3;;
loop
[000005] k1 = 0X1; xr3 = r2 + r4; yr5 = r6 + r7; yr8 = r9 * r10;;
[000009] j2 = j2 - 0X1; xr9 = r2 + r4; yr12 = r6 + r7; yr13 = r9 * r10;;
[00000D] IF njle, JUMP loop; k1 = 0X1; yr5 = r6 + r7; yr8 = r9 * r10;;
[000011] nop;;
[000012] nop;;
[000013] yr4 = 0X5;;
[000014] yr3 = 0X1;;
[000015] yr2 = 0;;
loop2
[000016] j10 = 0;;
[000017] yr3 = r3 - r2;;
[000018] yr4 = r4 - r3;;
[000019] IF nyaeq, JUMP loop2; nop;;
[00001B] nop;;
[00001C] nop; nop; nop;;
[00001F] xr5 = r5 + r5;;
[000020] j3 = 0X5;;
[000021] j2 = 0X1;;
[000022] k10 = 0X15;;
[000023] j10 = 0;;

Figure C-6. Using the Program Counter Value (ADSP-TS101 Processor)

Stepping

When single-step through a program, the simulator performs an instruction line step and skips invalid instructions (aborted, bubbles, or slots occupied by an invalid fetch).

Sometimes one step takes more than one cycle, and the **Pipeline Viewer** window advances by several lines, while the yellow arrow , which marks the current program counter location in the **Disassembly** window, moves to the next instruction line.

While skipping invalid instructions (aborted, bubbles, or an invalid fetch), the simulator still processes memory transactions. When the step is completed and the memory window is updated, a surprisingly large number of new values may appear. When debugging DMA, be aware that a single step may cause several DMA transactions to be completed.

Simulator Options

The **Simulator** submenu (under **Settings** menu) provides the **Configure DMA File I/O** command. This command opens the **DMA File I/O Configuration** dialog box (Figure C-7), from which to specify files as sources, destinations, or both for DMA transfers.

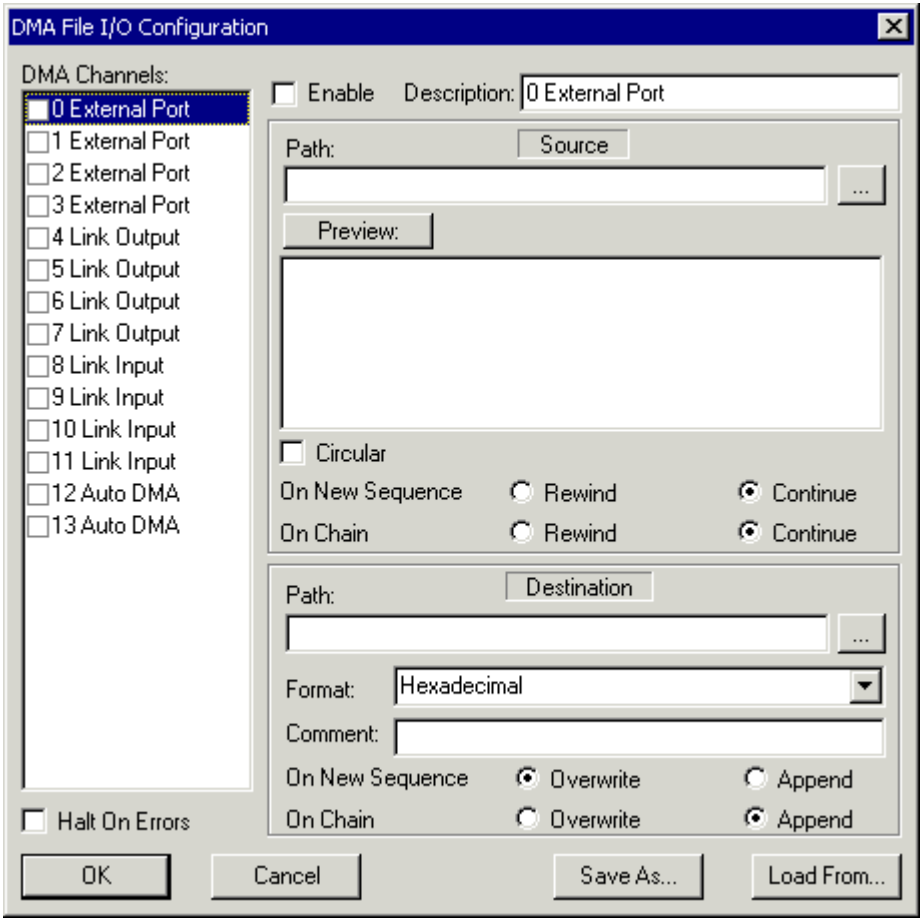

Figure C-7. DMA File I/O Configuration Dialog Box

For information about dialog box options and simulating a DMA transfer in the simulator, refer to VisualDSP++ online Help.

ADSP-TS20x Processors

This section includes the following topics, which apply to the ADSP-TS201, ADSP-TS202, and ADSP-TS203 processors.

- Simulator Timing Analysis Overview
- Pipeline Stages
- Stalls
- Aborts
- Pipeline Viewer and Disassembly Window Operations
- Simulator Options

Simulator Timing Analysis Overview

The ADSP-TS20x simulator is a cycle-accurate simulator which not only models the instruction set functionally, but also correctly models pipeline effects (stalls and aborts) and internal memory transactions timing.

Currently, the processor's external port and link ports are not modeled in a cycle-accurate manner. The simulator cycle-counts the code, but the cycle counts used for the external port or link ports are rough estimates of cycle counts that you could obtain by running the code on the chip. Do not rely on these counts for performance evaluation.

Use the **Pipeline Viewer** window to understand how processor timing affects the execution of your program. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-94 or refer to VisualDSP++ online Help.

Pipeline Stages

The ADSP-TS20x **Pipeline Viewer** window provides a representation of instruction flow through the processor's pipeline. Table C-2 lists the pipeline stages.

Table C-2. Pipeline Stages – ADSP-TS20x Processors

Stage	Abbreviation in Pipeline Viewer
Instruction Fetch 1	F1
Instruction Fetch 2	F2
Instruction Fetch 3	F3
Instruction Fetch 4	F4
Predecode	PD
Decode	D
Integer	I
Access	A
Execute Stage 1	EX1
Execute Stage 2	EX2

Stages `F1` through `F4` represent the fetch pipe. Stages `PD` through `EX2` represent the execution pipe. Fetch pipe stages contain raw quads of 32-bit words fetched from memory, not valid instruction lines. When you view the **Pipeline Viewer** window in disassembly format (by using the right-click menu), instructions that appear in the fetch pipe are not valid instruction lines, but merely bits and pieces. Valid instruction lines appear in the execution pipe stages.

Stalls

The examples that follow illustrate the way that Pipeline Viewer displays different types of stall events for ADSP-TS20x processors. For a complete list of pipeline effects and memory transaction timing, refer to the processor's hardware specification.

Stalls Due to IALU Dependency

The following sequence of instructions causes a 4-cycle stall at the PreDecode stage.

```
J0 = 0x40000;;
J1 = j31 + j0;;    //  This instruction is stalled in PreDecode
                   //  until the previous instruction reaches E2:
```

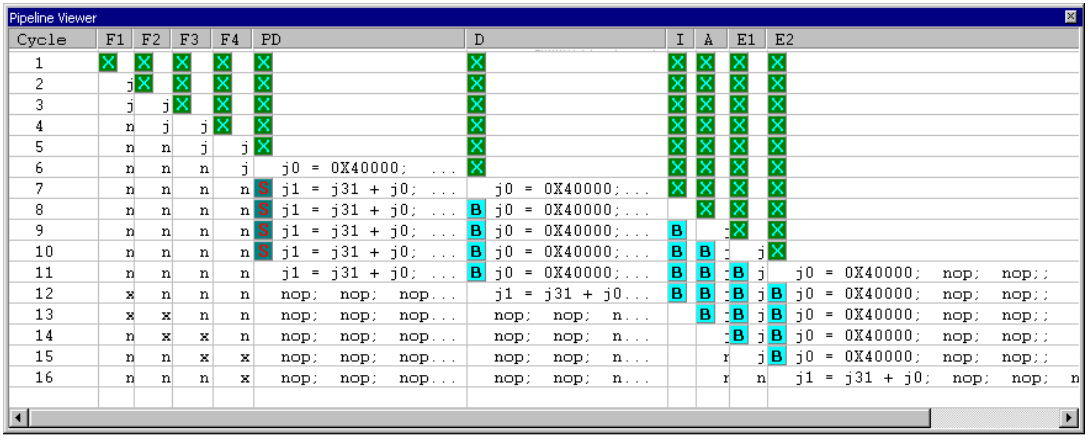

Figure C-8. Stall Due to IALU Dependency

Stalls Due to Compute Block Dependency

The following sequence of instructions causes a one-cycle stall at the Decode stage.

```
xr2 = r0 + r1;;
xr3 = r0 + r2;;    //  This instruction is stalled in Decode for
                   //  1 cycle. The source register (R2) is a
                   //  destination in the previous instruction.
```

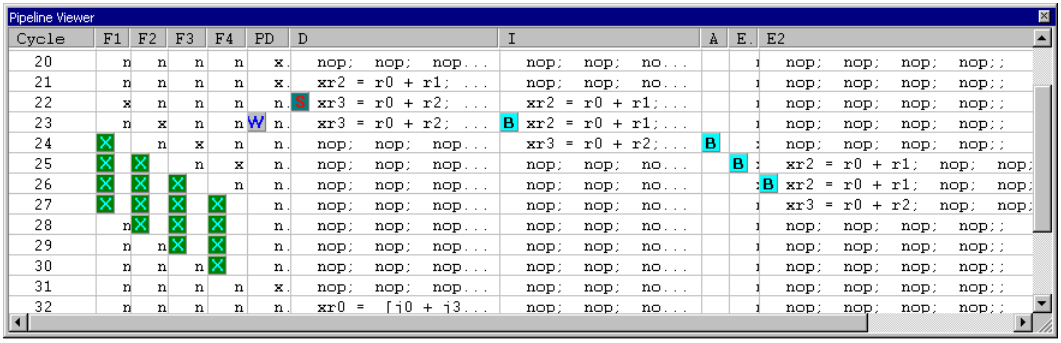

Figure C-9. Stall Due to Compute Block Dependency

Stalls Due to a Cache Miss

The following register load transaction causes a six-cycle stall because of a cache miss.

```
xr0=[j0 + j31];;
```

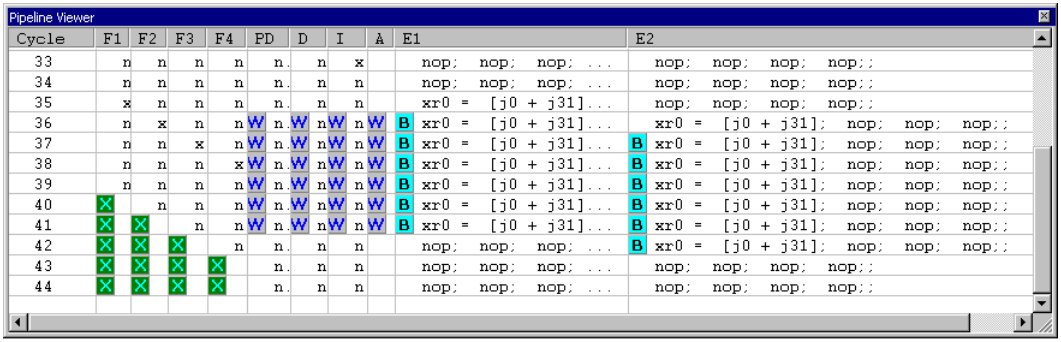
Pipeline Viewer
Cycle F1 F2 F3 F4 PD D I A E1 E2
33 n n n n n. n x nop; nop; nop; ... nop; nop; nop; nop;;
34 n n n n n. n n nop; nop; nop; ... nop; nop; nop; nop;;
35 x n n n n. n n xr0 = [j0 + j31]... nop; nop; nop; nop;;
36 n x n nW n.W nW nW B xr0 = [j0 + j31]... xr0 = [j0 + j31]; nop; nop; nop;;
37 n n x nW n.W nW nW B xr0 = [j0 + j31]... B xr0 = [j0 + j31]; nop; nop; nop;;
38 n n n xW n.W nW nW B xr0 = [j0 + j31]... B xr0 = [j0 + j31]; nop; nop; nop;;
39 n n n nW n.W nW nW B xr0 = [j0 + j31]... B xr0 = [j0 + j31]; nop; nop; nop;;
40 X n n nW n.W nW nW B xr0 = [j0 + j31]... B xr0 = [j0 + j31]; nop; nop; nop;;
41 X X n nW n.W nW nW B xr0 = [j0 + j31]... B xr0 = [j0 + j31]; nop; nop; nop;;
42 X X X n n. n n nop; nop; nop; ... B xr0 = [j0 + j31]; nop; nop; nop;;
43 X X X X n. n n nop; nop; nop; ... nop; nop; nop; nop;;
44 X X X X n. n n nop; nop; nop; ... nop; nop; nop; nop;;

Figure C-10. Stall Due to a Cache Miss

Aborts

The examples that follow illustrate how the Pipeline Viewer displays different types of abort events for ADSP-TS20x processors. For a complete list of pipeline effects and memory transaction timing, refer to the processor's hardware specification.

Aborts Due to an Unpredicted Change of Flow

In the following example, the abort at the Predecode stage is due to an unpredicted jump, predicated upon an IALU condition. The jump is unpredicted because the Branch Target Buffer is disabled in this particular example.

Aborted stages in the fetch pipe are marked with X, and aborted instructions in the execution pipe are marked with A .

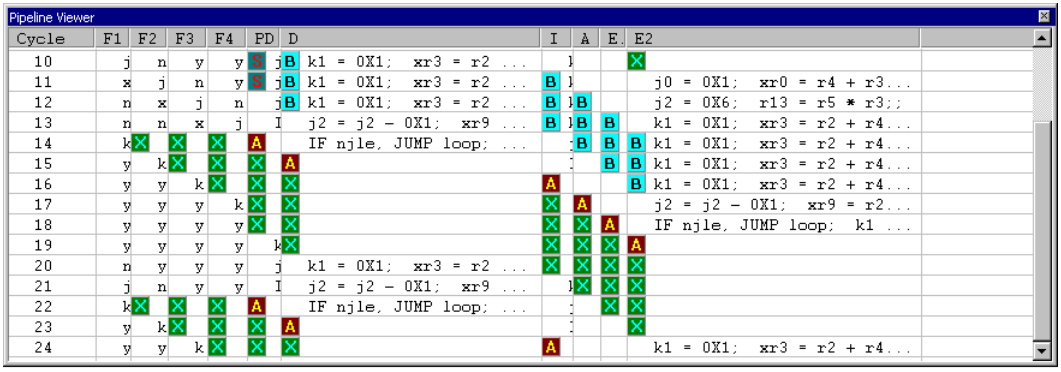
Pipeline Viewer
Cycle F1 F2 F3 F4 PD D I A E E2
10 k1 = 0X1; xr3 = r2 ...
11 k1 = 0X1; xr3 = r2 ... j0 = 0X1; xr0 = r4 + r3...
12 k1 = 0X1; xr3 = r2 ... j2 = 0X6; r13 = r5 * r3;;
13 j2 = j2 - 0X1; xr9 ... k1 = 0X1; xr3 = r2 + r4...
14 IF njle, JUMP loop; ... k1 = 0X1; xr3 = r2 + r4...
15 k1 = 0X1; xr3 = r2 + r4...
16 k1 = 0X1; xr3 = r2 + r4...
17 j2 = j2 - 0X1; xr9 = r2...
18 IF njle, JUMP loop; k1 ...
19
20 k1 = 0X1; xr3 = r2 ...
21 j2 = j2 - 0X1; xr9 ...
22 IF njle, JUMP loop; ...
23
24 k1 = 0X1; xr3 = r2 + r4...

Figure C-11. Abort Due to an Unpredicted Change of Flow

Abort Due to Mispredicted Change of Flow

In the following example, an abort appears at stage `E2` because of a mispredicted change of flow at the end of a loop, predicated on a compute block condition.

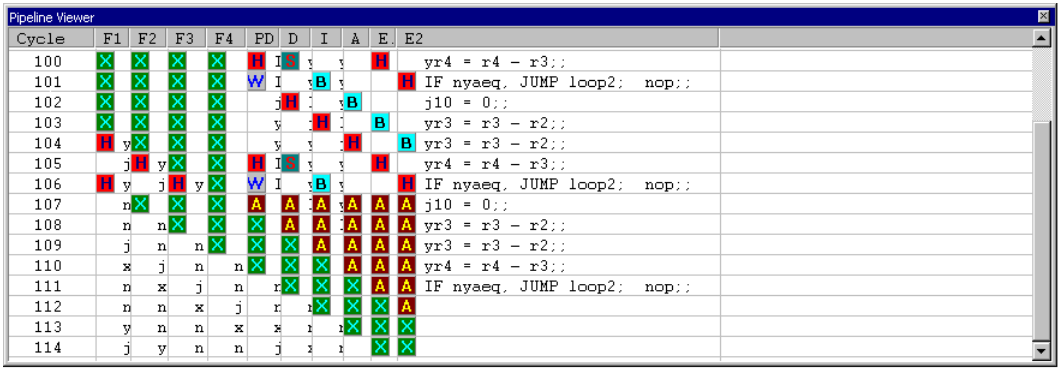

Figure C-12. Abort Due to Mispredicted Change of Flow

Branch Target Buffer Hits

In the following example, the first iteration of the loop causes a four-cycle penalty because of the unpredicted change of flow. On the second iteration, the change of flow instruction was found in the Branch Target Buffer and is marked with H. The branch is predicted and taken, and no penalty is exacted for the change of flow.

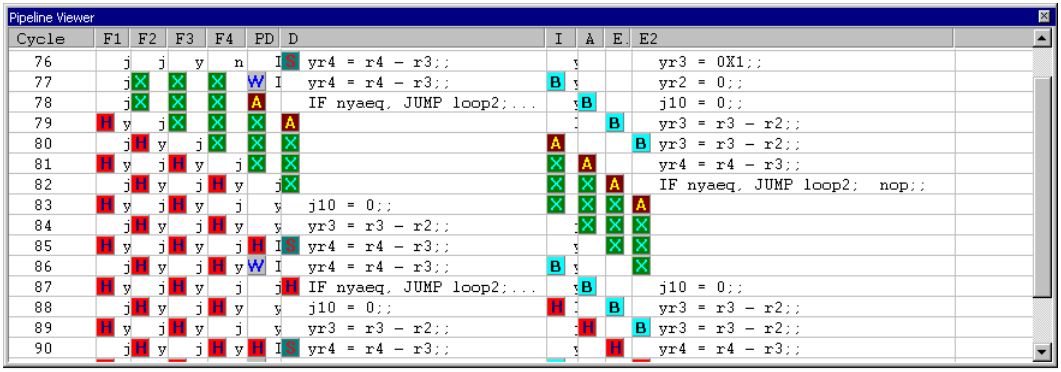
Pipeline Viewer
Cycle F1 F2 F3 F4 PD D I A E. E2
76 yr4 = r4 – r3;; yr3 = 0X1;;
77 yr4 = r4 – r3;; yr2 = 0;;
78 IF nyaeq, JUMP loop2;... j10 = 0;;
79 yr3 = r3 – r2;;
80 yr3 = r3 – r2;;
81 yr4 = r4 – r3;;
82 IF nyaeq, JUMP loop2; nop;;
83 j10 = 0;;
84 yr3 = r3 – r2;;
85 yr4 = r4 – r3;;
86 yr4 = r4 – r3;;
87 IF nyaeq, JUMP loop2;... j10 = 0;;
88 yr3 = r3 – r2;;
89 yr3 = r3 – r2;;
90 yr4 = r4 – r3;;

Figure C-13. Branch Target Buffer Hits

Pipeline Viewer and Disassembly Window Operations

This section includes the following topics.

- Current Program Counter Value
- Stepping

Current Program Counter Value

The program counter (PC) value, marked by ➪ in the **Disassembly** window, is the address of the instruction at stage `E2`. When a breakpoint is set at a certain address, the simulator halts after the instruction at this address is executed at stage `E2`.

You can specify which instruction the simulator executes next by manually changing the program counter value in the **PC Register** window. Note, however, that the current instruction, indicated by ➪ in the **Disassembly** window, is executed prior to the user-specified instruction. When you change a PC value, the simulator automatically flushes the pipeline and aborts all its instructions. After the pipeline is flushed, normal execution resumes from a memory address indicated by the new PC value. The simulator continues to run until an instruction at the new PC reaches stage `E2` of the pipeline.

Manually changing the PC value during simulation can result in unpredictable program behavior. VisualDSP++ does not safeguard against invalid PC values. Ensure that the specified PC indicates a valid instruction within program address space.

An example of an invalid PC is a data memory address whose contents is not recognized as an instruction by the simulator. In this case, the simulator generates an unhandled software exception.

Another example of an erroneous PC is the middle of an instruction line. Only part of the instruction line is executed, while instructions in the beginning of the line are ignored.

Figure C-14 shows how the program counter value is used for ADSP-TS20x processors.

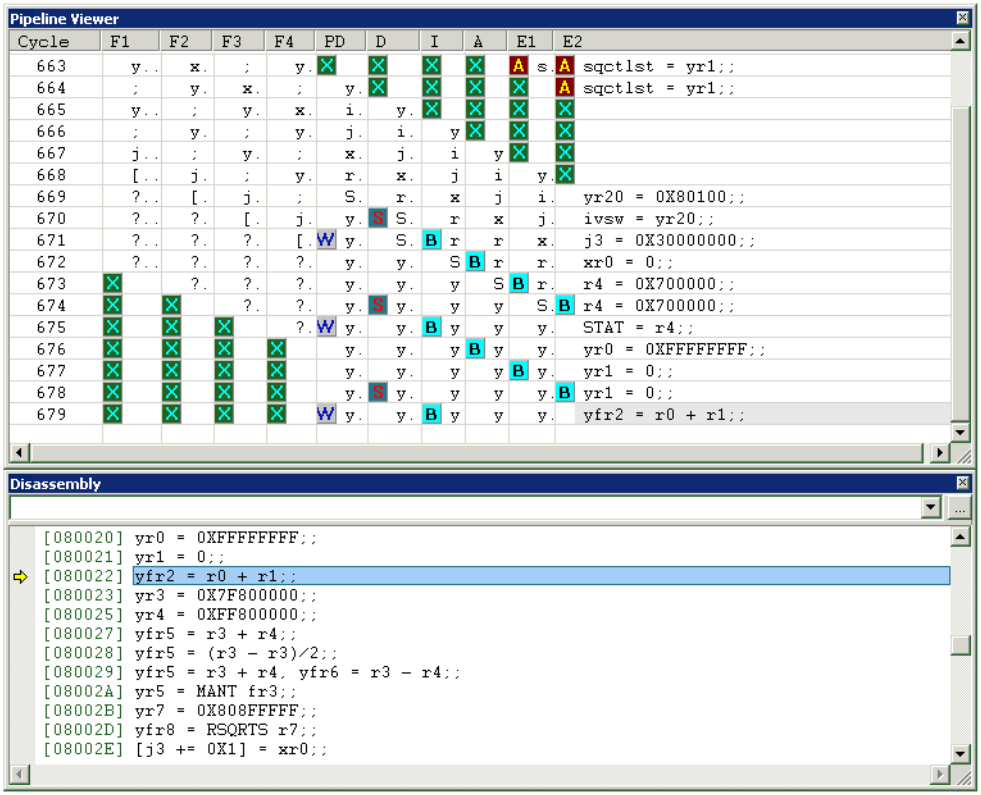

Figure C-14. Using the Program Counter Value (ADSP-TS20x Processors)

Stepping

When single-stepping through a program, the simulator performs an instruction line step and skips invalid instructions (aborted, bubbles, or slots occupied by an invalid fetch).

Sometimes one step takes more than one cycle, and the **Pipeline Viewer** window advances by several lines, while the yellow arrow , which marks the current program counter location in the **Disassembly** window, moves to the next instruction line.

While skipping invalid instructions (aborted, bubbles, or an invalid fetch), the simulator still processes memory transactions. When the step is completed and the memory window is updated, a surprisingly large number of new values may appear. When debugging DMA, be aware that a single step may cause several DMA transactions to be completed.

Simulator Options

The **Simulator** submenu under **Settings** provides the **Select Loader Program** command. This command opens the **Open File** dialog box, from which to specify a custom loader program. Once selected, the loader program automatically runs before a user program is loaded. The simulator defaults to a standard loader program (`TS20x_prom.dxe`, where `x` is 1, 2, or 3), but you can define your own loader by compiling a program into a `.DXE` file. If you create your own loader, your code must contain the label `_init_debug_end` to ensure that the loader is executed.

D SIMULATION OF BLACKFIN PROCESSORS

This appendix provides Blackfin simulator-specific information.

The information is organized as follows:

- "Peripheral Support in Simulators" on page D-2

  Note that VisualDSP++ online Help includes the most recent information about Blackfin processor simulation support.
- "Special Considerations for Peripherals" on page D-7
- "Simulator Instruction Timing Analysis for ADSP-BF535 Processors" on page D-9
- "Simulator Instruction Timing Analysis for ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 Processors" on page D-19
- "Multicycle Instructions and Latencies" on page D-22
- "Compiled Simulation" on page D-43

Peripheral Support in Simulators

Use the following key for the tables in this section.

Table D-1.

Symbol	
✔	Implemented
NA	Not applicable
NP	Not planned for implementation
FR	Planned for a future release

Table D-2 summarizes peripheral support in the ADSP-BF535 simulator.

Table D-2. Peripheral Support in the ADSP-BF535 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✔	✔	NA
UART	✔	✔	NA
PCI	NP	NP	NP
USB	NP	NP	NP
Flags	✔	✔	NA
System Timers	✔	✔	NA
RTC	✔	NA	NA
EBIU	NP	NA	NA

Table D-2. Peripheral Support in the ADSP-BF535 Simulator (Cont'd)

Peripheral Support	Modeled	Streamable	Bootable
SPI	✔	✔	NP
Watch Unit	NP	NA	NA
Trace Unit	NP	NA	NA
Core Timer	✔	✔	NA
MEMDMA	✔	✔	NA
PROM	FR	NA	FR

Table D-3 summarizes peripheral support in the ADSP-BF535 compiled simulator.

Table D-3. Peripheral Support in the ADSP-BF535 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✔	✔	FR
UART	FR	FR	FR
PCI	NP	NP	NP
USB	NP	NP	NP
Flags	✔	✔	NA
System Timers	FR	FR	NA
RTC	FR	NA	NA
EBIU	NP		NA
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA

Table D-3. Peripheral Support in the ADSP-BF535 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
Core Timer		FR	NA
MEMDMA	✔	✔	NA
PROM	NP	NA	NP

Table D-4 summarizes peripheral support in the ADSP-BF533 simulator.

Table D-4. Peripheral Support in the ADSP-BF533 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✔	✔	FR
UART	FR	FR	FR
Flags	FR	FR	NA
System Timers	✔	✔	NA
RTC	FR	NA	NA
EBIU	FR	NA	NA
PPI	✔	✔	NP
SPI	FR	FR	NP
Watch Unit	✔	NA	NA
Trace Unit	✔	NA	NA
Core Timer	✔	FR	NA
Watch Dog Timer	FR	NA	NA
PROM	NP	NA	NP

Table D-5 summarizes peripheral support in the ADSP-BF533 compiled simulator.

Table D-6 summarizes peripheral support in the ADSP-BF561 simulator.

Table D-5. Peripheral Support in the ADSP-BF533 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	FR	FR	FR
UART	FR	FR	FR
Flags	NP	NP	NA
System Timers	FR	FR	NA
RTC	NP	NP	NA
EBIU	NP	NP	NA
PPI	FR	FR	NP
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA
Core Timer	✔	FR	NA
Watch Dog Timer	FR	NA	NA
PROM	NP	NA	FR

Table D-6. Peripheral Support in the ADSP-BF561 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	FR	FR	FR
UART	FR	FR	FR
Flags	FR	FR	NA
System Timers	✔	✔	NA
RTC	FR	NA	NA
EBIU	FR	NA	NA
PPI	✔	✔	NP

Table D-6. Peripheral Support in the ADSP-BF561 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA
Core Timer	✔	FR	NA
Watch Dog Timer	FR	NA	NA
PROM	FR	NA	FR

Special Considerations for Peripherals

This section describes the limitations of the simulation software models.

Universal Asynchronous Receiver/Transmitter Peripheral

You can manipulate all the UART configuration bits. Currently, you cannot simulate the data error (Framing Error, Parity Error, Break Interrupt) conditions or the **Modem Status** register status bits (Data Carrier Detect, Ring Indicator, Data Set Ready, Clear To Send). You can specify **Set Break** in the Line Control register, but this setting has no effect. The current simulator does not model the `IRCR` register.

Timer (TMR) Peripheral

In Width Capture (`WDTH_CAP`) mode, the timer counts the number of clocks in both the width and period. The waveform that the timer reads is attached via the **Streams** dialog box in VisualDSP++.

You can attach a file to the following device names.

- TIMER0_WDTH_CAP
- TIMER1_WDTH_CAP
- TIMER2_WDTH_CAP

The format of the input file is as follows.

```
PERIOD_COUNT
WIDTH_COUNT
PERIOD_COUNT
WIDTH_COUNT
```

In WDTH_CAP mode, the timer reads two 32-bit values from the input file. The first value is the number of pulses (clocks) in the period. The second value is the number of pulses in the width.

When PULSE_HI is set, the timer delivers high widths and low periods. When PULSE_HI is not set, the timer delivers low widths and high periods.

Simulator Instruction Timing Analysis for ADSP-BF535 Processors

The ADSP-BF535 Family Simulator is a core cycle-accurate simulator with an eight-stage pipeline. The simulator models all the sequencer and memory events of the ADSP-BF535 processor.

The Pipeline Viewer enables you to understand the execution timing of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-94 or VisualDSP++ online Help.

The Pipeline Viewer for the ADSP-BF535 processor displays stages Decode through Writeback. The first two stages of the pipeline, IF1 and IF2, are not displayed because the information, provided by the simulator, in those stages is not significant.

Stall Reasons

The stall reasons are grouped in three categories:

- Multicycle instructions latencies (see "Multicycle Instructions and Latencies" on page D-22)
- Instructions latencies (see "Instruction Latencies" on page D-26)
- L1 data memory latencies (see "L1 Data Memory Stalls" on page D-34)

They are reported in the Pipeline Viewer as:

- Data address generator (DAG) read-after-write (RAW) hazard
- Data register (dreg) hazard: two cycle
- Dreg register (dreg) hazard: one cycle

- Memory stall
- Memory-mapped register (MMR) stall
- `CSYNC` stall
- `SSYNC` or IDLE `SYNC` stall
- Raise stall
- Single-step (SS) mode
- RET read after write
- Unidentified stall

Kill Reasons

The kill reasons are as follows.

- Branch Kill – change of flow
- Mispredict – mispredicted conditional change of flow
- **Refetch** – refetch, such as following an `IDLE` instruction
- Interrupt – interrupt/exception

Pipeline Viewer Window Examples

Figure D-1 shows a RAW hazard stall.

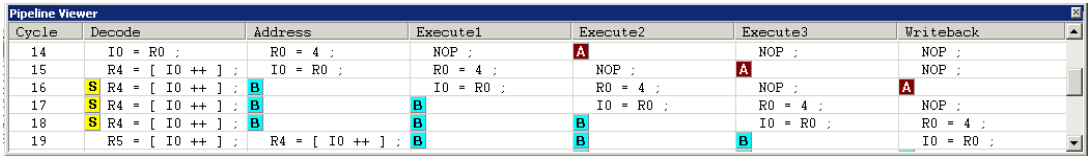
Pipeline Viewer
Cycle
Decode
Address
Execute1
Execute2
Execute3
Writeback
14 I0 = R0 ; R0 = 4 ; NOP ; NOP ; NOP ;
15 R4 = [I0 ++] ; I0 = R0 ; R0 = 4 ; NOP ; NOP ;
16 R4 = [I0 ++] ; I0 = R0 ; R0 = 4 ; NOP ;
17 R4 = [I0 ++] ; I0 = R0 ; R0 = 4 ;
18 R4 = [I0 ++] ; I0 = R0 ; R0 = 4 ;
19 R5 = [I0 ++] ; R4 = [I0 ++] ; I0 = R0 ;

Figure D-1. RAW Hazard Stall

These stalls are detected in the Decode stage. The instruction stalls there until all DAG registers required and updated in later pipe stages are available.

In this example, the instruction "`I0 = R0;`" in the Execute1 (cycle 16), Execute2 (cycle 17) and Execute3 (cycle 18) stage is stalling the instruction "`R4 = [ I0++ ];`" in the Decode stage. This stall is caused by the first instruction because it updates the value of `I0` in stage Writeback, while the second instruction needs the value of `I0` in the Address stage to increment `I0`.

Figure D-2 shows a fetch stall.

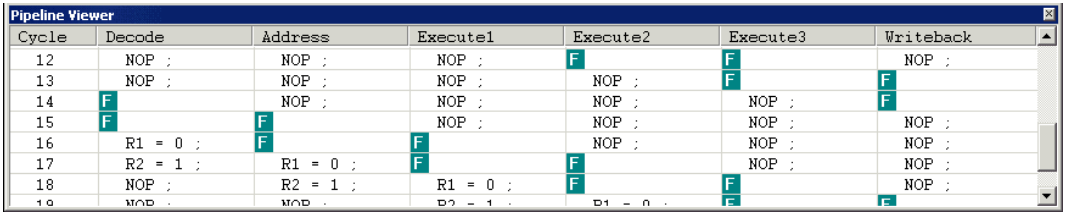
Pipeline Viewer
Cycle
Decode
Address
Execute1
Execute2
Execute3
Writeback
12 NOP ; NOP ; NOP ; NOP ;
13 NOP ; NOP ; NOP ; NOP ;
14 NOP ; NOP ; NOP ; NOP ;
15 NOP ; NOP ; NOP ;
16 R1 = 0 ; NOP ; NOP ;
17 R2 = 1 ; R1 = 0 ; NOP ;
18 NOP ; R2 = 1 ; R1 = 0 ; NOP ;

Figure D-2. Fetch Stall

Fetch stalls are detected in the Decode stage and are caused by memory latencies when an instruction is fetched.

In this example, two fetch stalls appear in the Decode stage (cycles 14 and 15) because of a memory latency when instruction "`R1 = 0;`" is fetched. These fetch latencies are then propagated in the pipeline: stage "Address" (cycles 15 and 16), stage "Execute 1" (cycles 16 and 17), and so on.

Pipeline Viewer Window Messages

When you hold the **Ctrl** key down and pause the mouse over a pipeline viewer event icon indicating instructions, the **Pipeline Viewer** window displays informational messages. An example is shown in Figure D-4 on page D-20.

These types of messages may appear:

- Stalls detected
- Kills detected
- Multicycle instruction messages

Pipeline Viewer Detail View Stall Event Messages

Table D-7 shows the messages that occur when a stall is detected.

Table D-7. Stalls Detected Messages (ADSP-BF535 Processor)

Message	Explanation	Example
ICache miss	Instruction cache miss	
IAU empty	Instruction alignment unit empty	
DCache miss	Data cache miss	
DCache store buffer full	Data cache buffer overflow. The processor stalls until the FIFO moves forward and a space is free.	

Table D-7. Stalls Detected Messages (ADSP-BF535 Processor) (Cont'd)

Message	Explanation	Example
DCache load while store pending	A load access collides with a pending store access in the store buffer. (They are tying to access the same address.)	
DCache load while store pending w/ size mismatch	Load access size is different from that of the store access. The buffer must be flushed before the load can be carried out.	
DCache bank collision	The addresses in a dual- memory access command are accessing the same minibank. It does not matter whether both are loads, or load and store.	
SYNC with store pending	SYNC instructions force all speculative, transient in the core/system to be completed before proceeding.	`SSYNC;`
EU->MUL/MAC RAW hazard	Execution unit, Multiply or Multiply accumulate with a read after write hazard	`R0 = R1 + R0;` `P0 = R0;`
RETx RAW hazard	Writing to one of the RETx (RETS, RETI, RETX, RETN, or RETE) registers immediately followed by the corresponding return instructions.	`RETX = R0;` `RTX;`
Dagreg WAW hazard	Writing to one of the DAG registers, and immediately writing to it again.	`I3 = R3;` `I3 += M0;`
Dagreg RAW hazard	Writing to one of the DAG registers, and immediately reading	`I3 = R3;` `[I3] = R7;`
dsp32alu implied ireg dependency RAW hazard		
ccMV preg->dreg RAW hazard	A conditional move of a preg into a dreg, followed by a read of the dreg	`If CC R0 = P1;` `R0 = R1;`
ccMV dreg->dreg RAW hazard	A conditional move of a dreg into a dreg, followed by a read of the source dreg	`If CC R0 = R1;` `R2 = R0;`
ccMV dpreg->preg RAW hazard	A conditional move of a dreg into a preg, followed by a read of the preg	`If CC P0 = R1;` `P1 = P0 ;`

Table D-7. Stalls Detected Messages (ADSP-BF535 Processor) (Cont'd)

Message	Explanation	Example
loopsetup WAW hazard	A LSETUP instruction followed by another LSETUP, both writing to the same LC reg	`LSETUP (LS,LE)LC0=P0;` `LSETUP (LS,LE)LC0=P1;`
loopsetup while lc is nonzero	Using an LSETUP instruction and writing a value other than zero to the Lcreg	`LSETUP (LS,LE)LC0=P0;` `Nop;`
loop top/bot RAW hazard	Writing to a loop top/bottom register, followed by a read of the same register	`LT0 = R0;` `R2 = LT0;`
write to loop cnt stall	A write to a LCreg, followed by any op	`LC0 = R0;` `Nop; (`*any op*`)`
multicycle ALU2op instruction	A two-operand ALU instruction requiring more than one cycle to complete	`R0 *= R1;`
multicycle DAG instruction		`[--SP] =` `(R7:0,P5:0);`
CC2dreg RAW hazard	Reading the CC register into a dreg, and then reading that register	`R0 = CC;` `CC = R0;`
Mac/video after regmv sysreg to dreg raw hazard	Register move of a system register to a dreg, followed by a MAC or video instruction	`R0 = LC0;` `R2.H = R1.L * R0.H;`
Regmv sysreg to dreg followed by ALU op dreg raw hazard	Writing a system register to a dreg, followed by an ALU operation using that dreg as an operand	`R0 = LC0;` `R2 = R1 + R0;`
Video after extracted 3-input add dreg raw hazard		
Extracted 3-input add followed by special dsp32 instruction		
Search followed by exu operation dreg raw hazard	A search instruction followed by any execution instruction with an operand of a dreg used in the search instruction	`(R3,R0) = search R1 (LE);` `R2.H =R1.L * R0.H;`

Table D-7. Stalls Detected Messages (ADSP-BF535 Processor) (Cont'd)

Message	Explanation	Example
Regmv hazard: preg to dreg -> dreg to sys/preg RAW	A register move of a preg to a dreg, followed by another register move of that same dreg to a system register or preg	`R0 = P0;` `ASTAT = R0;`
Regmv hazard: sysreg to dreg -> dreg to dreg RAW	A register move of a system register to a dreg, followed by another register move of that same dreg to a dreg	`R0 = ASTAT;` `R1 = R0;`
Regmv hazard: sysreg to dreg -> dreg to sysreg RAW	A register move of a system register to a dreg, followed by another register move of that same dreg to a system register	`R0 = LC0;` `ASTAT = R0;`
Regmv hazard: sysreg to areg -> dreg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of a dreg to the same accumulator register	`A0.w = LC0;` `A0 =R0;`
Regmv hazard: sysreg to areg -> preg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of a preg to that same accumulator register	`A0.w = LC0;` `A0 =P0;`
Regmv hazard: sysreg to areg -> areg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of an accumulator register to that same accumulator register	`A0.w = LC0;` `A0 =A1;`
Regmv hazard: sysreg to areg -> areg to dreg RAW	A register move of a system register to an accumulator register, followed by another register move of that same accumulator register to a dreg	`A0.w = LC0;` `R0 =A0;`
Regmv hazard: sysreg to areg -> areg to sysreg RAW	A register move of a system register to an accumulator register, followed by another register move of that same accumulator register to a system register	`A0.w = LC0;` `ASTAT = A0.w;`
Regmv hazard: sysreg to areg -> load to areg WAW	A register move of a system register to an accumulator register, followed by a load to the same accumulator register	`A0.w = LC0;` `A0.w = [I0];`
Regmv hazard: sysreg to areg -> exu op using areg RAW	A register move of a system register to an accumulator register, followed by any execution unit operation using that accumulator register as an operand	`A0.w = LC0;` `A0 = A0(S);`

Table D-7. Stalls Detected Messages (ADSP-BF535 Processor) (Cont'd)

Message	Explanation	Example
AQreg hazard: move to AQ -> exu op using AQ RAW		
CCreg hazard: move to CC -> exu op using CC RAW		

Kills Detected Messages

Table D-8 shows the messages that occur when a kill is detected.

Table D-8. Kills Detected Messages (ADSP-BF535 Processor)

Message	Explanation	Example
change-of-flow kill	A branch	`CALL (P0);`
rti change-of-flow kill	Return from interrupt kills	`RTI;`
mispredicted change-of-flow kill	Kills due to mispredicted branches	`R0 = 0; CC = R0;` `If CC JUMP next (bp);`
hardware loop bottom kill		
interrupt kill	Instructions in the pipeline are killed due to an interrupt	`RAISE 1`
sync kill	SYNC instructions force all speculative, transient in the core/system to be completed before proceeding, killing instructions in the pipe	`SSYNC;`

Multicycle Instructions

Multicycle instructions are a category of instructions that cannot be completed in less than two cycles. Consequently, the extra cycles generated by such an instruction cannot be removed without removing the multicycle instruction itself.

In Figure D-3, multicycle instruction "`[--SP] = (R7:6, P5:3)`" enters the pipeline Decode stage at cycle 16 and takes five cycles to complete (1 cycle per register to push on the stack SP). The next instruction "`R7 = 0`" takes only one cycle.

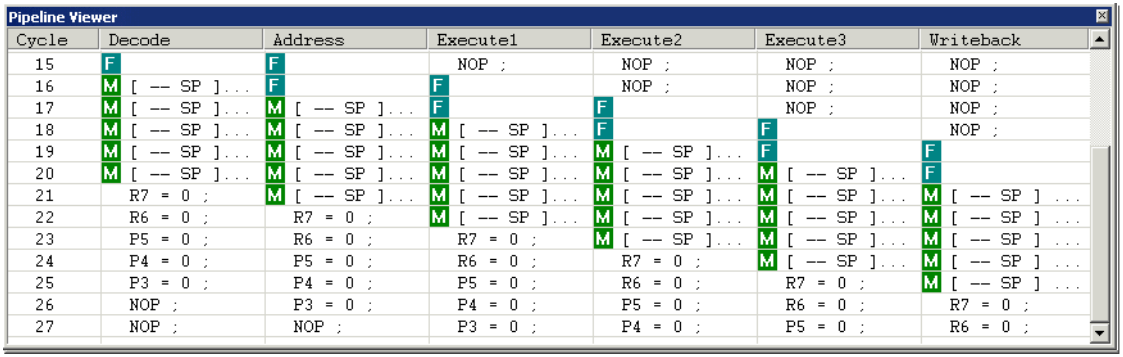
Pipeline Viewer
Cycle Decode Address Execute1 Execute2 Execute3 Writeback
15 F NOP ; NOP ; NOP ; NOP ;
16 M [-- SP]... F F NOP ; NOP ; NOP ; NOP ;
17 M [-- SP]... M [-- SP]... F F NOP ; NOP ;
18 M [-- SP]... M [-- SP]... M [-- SP]... F F NOP ;
19 M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP]... F F
20 M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP]... F
21 R7 = 0 ; M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP] ...
22 R6 = 0 ; R7 = 0 ; M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP] ...
23 P5 = 0 ; R6 = 0 ; R7 = 0 ; M [-- SP]... M [-- SP]... M [-- SP]... M [-- SP] ...
24 P4 = 0 ; P5 = 0 ; R6 = 0 ; R7 = 0 ; M [-- SP]... M [-- SP] ...
25 P3 = 0 ; P4 = 0 ; P5 = 0 ; R6 = 0 ; R7 = 0 ; M [-- SP] ...
26 NOP ; P3 = 0 ; P4 = 0 ; P5 = 0 ; R6 = 0 ; R7 = 0 ;
27 NOP ; NOP ; P3 = 0 ; P4 = 0 ; P5 = 0 ; R6 = 0 ;

Figure D-3. Example of a Multicycle Instruction in the Pipeline Viewer

For details about multicycle instructions, see "Multicycle Instructions and Latencies" on page D-22.

Abbreviations in Pipeline Viewer Messages

Table D-9 shows abbreviations that may appear in the **Pipeline Viewer** window.

Table D-9. Abbreviations in the Pipeline Viewer Window

Abbreviation	Meaning
ALU	Arithmetic Logic Unit operations (Logical ops, Bit ops, Shift/Rotate ops, Arithmetic ops excluding Mult, Vector ops excluding Mult/MAC)
ALU2op	A two-operand ALU instruction
AQreg	
CC2dreg	CC register move to a dreg
ccMV	Conditional move
CCreg	CC register. This multipurpose flag typically holds the result of an arithmetic comparison.
DAG	Data Address Generator unit
Dagreg	A DAG register (for example, P5-0, I3-0, M3-0, B3-0, and L3-0)
dreg	Data register (for example, R7-0 or A1-0)
Dsp32alu	A 32-bit DSP ALU instruction
EXU	Execution unit
IAU	Instruction Alignment Unit
MAC	Multiplier/Accumulator Unit
MUL	Multiplier Unit operations (for example, Vector Multiply, 32-bit Multiply, Vector MAC)
preg	Pointer register (for example, P5-0, FP, USP, or SSP)
RAW	Read after write
regmv	A register move
sysreg	System Register (for example, LC1/0, LB1/0, LT1/0, SYSCFG, SEQSTAT, ASTAT, RETS, RETI, RETX, RETN, RETE, CYCLES, and CYCLE2)
WAW	Write after write
Video	Video operations (video pixel operations)

Simulator Instruction Timing Analysis for ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 Processors

The simulator for the ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 processors is a core cycle-accurate simulator with a ten-stage pipeline. The simulator models all sequencer and memory events.

The Pipeline Viewer enables you to understand the execution timing of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-94 or VisualDSP++ online Help.

Stall Reasons

The stall reasons are as follows.

- Data address generator (DAG) read-after-write (RAW) hazard
- Memory stall
- Memory-mapped register (MMR) stall
- Unidentified stall
- Data register (dreg) hazard: two cycle
- Dreg hazard: one cycle
- `CSYNC` stall
- `SSYNC` or `IDLE SYNC` stall
- `LSETUP0` and not `LP0_ALLOWED`
- Awkward loop

- Raise stall
- SS mode
- RET read after write

Kill Reasons

The kill reasons are as follows.

- Branch Kill – change of flow
- Mispredict – mispredict conditional change of flow
- Interrupt – interrupt/exception
- **Refetch** – refetch, such as following an `IDLE` instruction

Pipeline Viewer Window Examples

Figure D-4 shows a RAW hazard stall.

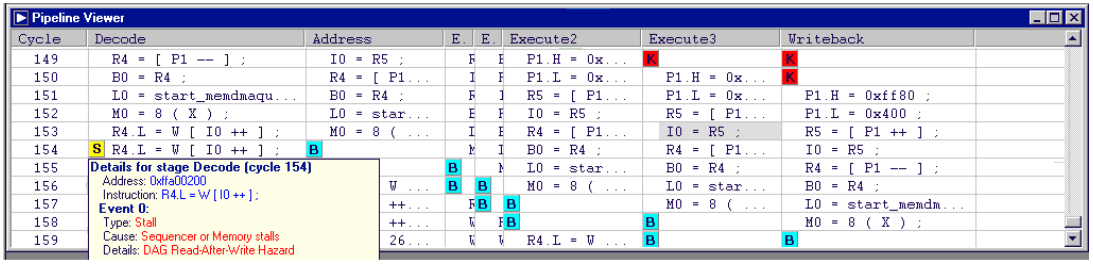
Pipeline Viewer
Cycle
Decode
Address
E.
E.
Execute2
Execute3
Writeback
149 R4 = [P1 --] ; I0 = R5 ; P1.H = 0x... P1.H = 0x...
150 B0 = R4 ; R4 = [P1... P1.L = 0x... P1.L = 0x...
151 L0 = start_memdmaqu... B0 = R4 ; R5 = [P1... P1.H = 0xff80 ;
152 M0 = 8 (X) ; L0 = star... I0 = R5 ; R5 = [P1... P1.L = 0x400 ;
153 R4.L = W [I0 ++] ; M0 = 8 (... R4 = [P1... I0 = R5 ; R5 = [P1 ++] ;
154 R4.L = W [I0 ++] ; B0 = R4 ; R4 = [P1... I0 = R5 ;
155 L0 = star... B0 = R4 ; R4 = [P1 --] ;
156 M0 = 8 (... L0 = star... B0 = R4 ;
157 M0 = 8 (... L0 = start_memdm...
158 M0 = 8 (X) ;
159
Details for stage Decode (cycle 154)
Address: 0xffa00200
Instruction: R4.L = W [I0 ++] ;
Event 0:
Type: Stall
Cause: Sequencer or Memory stalls
Details: DAG Read-After-Write Hazard

Figure D-4. RAW Hazard Stall

These stalls are detected in the Decode stage. The instruction stalls until all DAG registers required and updated in later pipe stages are available.

In the example, `I0=R5` in the Execute3 stage is stalling the instruction in decode, which wants to increment I0.

Figure D-5 shows an MMR stall.

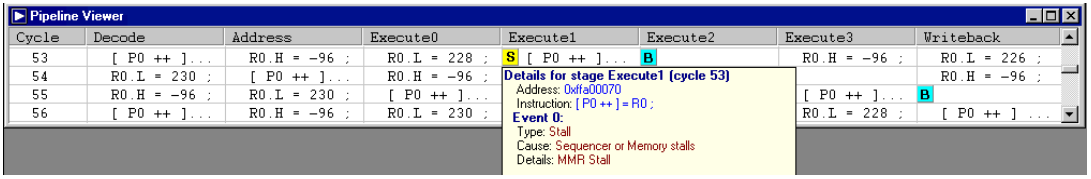

Figure D-5. MMR Stall

MMR stalls occur in E1 while the MMR value is being returned.

Figure D-6 shows a branch kill.

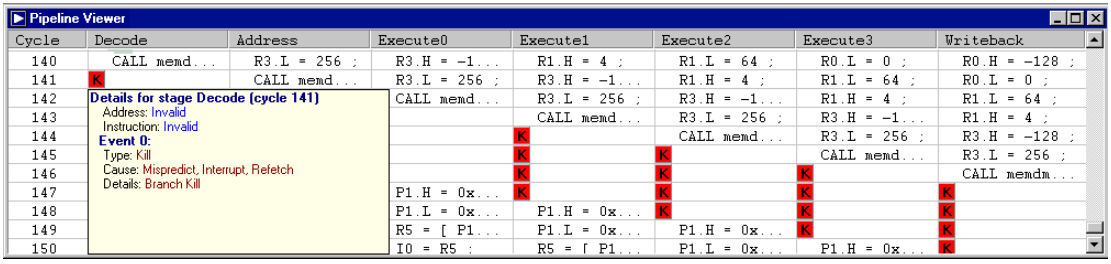

Figure D-6. Branch Kill

In this example, an unconditional control transfer kills several stages that were behind it. Fetching begins at the destination of the control transfer instruction after the killed stages.

Multicycle Instructions and Latencies

This section contains a description of all Blackfin processor multicycle instructions and latencies.

Multicycle behavior exists when an instruction, sometimes only under certain circumstances, is completed in more than one cycle. This cycle loss cannot be avoided without removing the instruction that caused it.

A *latency condition* exists when a pair of instructions incur extra cycles between them because of their proximity to each other in the code. You can avoid a latency condition's cycle loss by separating the two instructions by as many instructions as cycles lost. Each multicycle and latency entry indicates whether it is currently supported in the simulation environment.

All multicycle and latency conditions described here are native to the first implementation of Blackfin processor architecture. Future implementations may be different. The tables in this section show the cycle latencies of the 10x core processors, represented by the ADSP-BF532 and the ADSP-BF535 processor.

Multicycle Instructions

All instructions not mentioned here are completed in one cycle. This section describes instructions that take more than one cycle. Instruction names are consistent with the *Blackfin Processor Instruction Set Reference*. The cycle counts in the following examples represent the entire cycle time of the instruction shown.

Push Multiple or Pop Multiple

`PushPopMultiple` is completed in *n* cycles, where *n* is the number of registers pushed or popped.

Table D-10. PushPopMultiple Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
[--SP] = (R7:0,P5:0);	14 cycles	14 cycles
(R7:0,P5:3) = [SP++];	11 cycles	11 cycles

32-Bit Multiply (modulo 2^{32})

Table D-11. Bit Multiply Instruction and Cycles

Instruction	ADSP-BF532	ADSP-BF535
R0 *= R1;	3 cycles	5 cycles

Call and Jump

Table D-12. Call and Jump Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
CALL 0x22;	5 cycles	4 cycles
CALL (PC + P0);	5 cycles	4 cycles
CALL (P0);	5 cycles	4 cycles
JUMP 0x22;	5 cycles	4 cycles
JUMP (PC + P0);	5 cycles	4 cycles
JUMP (P0);	5 cycles	4 cycles

Conditional Branch

The number of cycles that a branch takes depends on the prediction as well as the actual outcome.

Table D-13. Conditional Branch Cycles

Prediction	taken				not taken			
Outcome	taken		not taken		taken		not taken	
Cycle Time	BF532	BF535	BF532	BF535	BF532	BF535	BF532	BF535
	4 cycles	4 cycles	8 cycles	7 cycles	8 cycles	7 cycles	1 cycle	1 cycle

Return

Table D-14. Return Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
RTX;	5 cycles	7 cycles[1]
RTE;	5 cycles	7 cycles[1]
RTN;	5 cycles	7 cycles[1]
RTI;	5 cycles	7 cycles
RTS;	5 cycles	4 cycles

1 Best case

Core and System Synchronization

Table D-15. Core and System Synchronization Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
CSYNC;	10 cycles	7 cycles
SSYNC;	10 cycles	7 cycles

Linkage

Table D-16. Linkage Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
LINK 8;	3 cycles	4 cycles
UNLINK;	2 cycles	3 cycles

Interrupts and Emulation

Table D-17. Interrupts and Emulation Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
RAISE 10;	3 cycles	3 cycles
EXCPT 3;	3 cycles	7 cycles
EMUEXCPT;	3 cycles	3 cycles[1]
STI R4;	3 cycles	3 cycles[1]

1 Best case as determined by physical characteristics of external memory

TESTSET

The `TESTSET` instruction is a multicycle instruction that is executed in a variable number of cycles. It is dependent on the cycles needed for a read acknowledge from off-core L2 memory and whether the address being tested is both in the cache and dirty. The number of cycles is determined as follows.

`cycles = 1 (instruction) + 1 (stall) +` *`x`* `(read ack) +` *`y`* `(cache penalty)`

In an optimal environment, *x* would be 5 and *y* would be zero. If the address resides in a dirty line, *y* is determined by the cycles to fill the dirty line plus any core boundary latencies. The address should not reside dirty

in the cache as the address contents are meant to be updated across multiple processors and not be a local variable. This instruction is dependent on off-core conditions, so it is not modeled by the simulation environment.

Table D-18. TESTSET Instruction

Instruction	ADSP-BF535
TESTSET (P0);	7+ cycles[1]

1 Best case as determined by physical characteristics of external memory

Instruction Latencies

In addition to being based on instruction, instruction latencies are contingent on placement of specific instruction pairs relative to one another. You can avoid them by separating them by as many instructions as the number of cycles incurred between them. For example, if a pair of instructions incur a 2-cycle latency, separating them by two instructions will eliminate that latency.

In the tables that follow note that bold type identifies register dependencies within the instruction pairs. No bold type in an entry means that the latency condition will occur regardless of what registers are used.

For a list of accumulator to data register (`Areg2Dreg`), math, video, multiply, and ALU ops, as well as register groupings, see "Instruction Groups" on page D-41 and "Register Groups" on page D-42. Instruction names are consistent with the *Blackfin Processor Instruction Set Reference*.

Calculate the total cycle time of each entry by adding the cycles taken by the instruction to the number of stall cycles for the instruction.

Accumulator to Data Register Latencies

Table D-19. Accumulator to Data Register Latencies

Description	Example <cycles + stalls > instruction	
	BF532	BF535
dreg = Areg2Dreg op video op using **dreg** as src	1 1 + 1	< 1 > **R1** = R6.L * R4.H (IS); < 1 + 2 > R5 = BYTEOP1P (R3:2, **R1**:0);
dreg = Areg2Dreg op rnd12/rnd20 using **dreg** as src	1 1	< 1 > **R4**.L = (A0 = R3.H*R1.H); < 1 + 1 > R0.H = R2 + **R4** (RND12);
dreg = Areg2Dreg op shift/rotate op using **dreg** as src	1 1	< 1 > **R4**.L = (A0 = R3.H*R1.H); < 1 + 1 > R1 = ROT R2 BY **R4**.L;
dreg = Areg2Dreg op add on sign using **dreg** as src	1 1	< 1 > **R0**.H=**R0**.L=SIGN(R2.H)*R3.H+SIGN(R2.L)*R3.L; < 1 + 1 > R6.H=R6.L= SIGN(**R0**.H)*R1.H+SIGN(**R0**.L)*R1.L;
dreg = math op Areg2Dreg op using **dreg** as src	1 1	< 1 > **R2** = R3 + R1; < 1 + 1 > R4.H = **R2**.L * R0.H;

Register Move Latencies

In each of the following cases the stall condition occurs when the same register is used in both instructions.

Table D-20. Register Move Latencies

Description	Example <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
dreg = sysreg ALU op using **dreg** as src (or vector ALU op)	1 1 1 1	< 1 > **R0** = LC0; < 1 + 1 > R2 = R1 + **R0**; < 1 > **R2** = LC0; < 1 + 1 > R1.L = **R2** (RND);
dreg = preg sysreg = **dreg**	1 1	< 1 > **R0** = P0; < 1 + 1 > ASTAT = **R0**;
dreg = sysreg dreg = **dreg**	1 1	< 1 > **R0** = ASTAT; < 1 + 1 > R1 = **R0**;
dreg = sysreg multiply/video op with **dreg** as src	1 1 + 1	< 1 > **R0** = LC0; < 1 + 2 > R2.H = R1.L * **R0**.H;
dreg = sysreg accreg = **dreg**	1 1	< 1 > **R0** = LC0; < 1 + 1 > A0 = **R0**;
preg = dreg any processor op using **preg**	1 1 + 4	< 1 > **P0** = R3; < 1 + 3 > R0 = **P0**;
dagreg = dreg any processor op using **dagreg**	1 1 + 4	< 1 > **I3** = R3; < 1 + 3 > R0 = **I3**;
dreg = sysreg sysreg = **dreg**	1 1	< 1 > **R0** = LC0; < 1 + 1 > ASTAT = **R0**;
accreg = sysreg **accreg** = dreg	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0** = R0;
accreg = sysreg **accreg** = preg	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0.w** = P0;
accreg = sysreg accreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > A1 = **A0**;

Table D-20. Register Move Latencies (Cont'd)

Description	Example <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
accreg = sysreg dreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > R0.L = **A0.x**;
accreg = sysreg sysreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > ASTAT = **A0.w**;
accreg = sysreg math op using **accreg** as src	1 1	< 1 > **A1.x** = LC0; < 1 + 1 > R1.H = (A0+=**A1**);
accreg = sysreg POP to **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0.w** = [SP ++];
POP to **dagreg** any processor op using **dagreg**	1 1 + 3	< 1 > **I3** = [SP++]; < 1 + 3 > R0 = **I3**;
LOAD/POP to **preg** any processor op using **preg**	1 1 + 3	< 1 > **P3** = [SP++]; < 1 + 3 > R0 = **P3**;
R0.L = R1.L+R2.L R3 = R0.H*R4.L	The 10x core considers register halves to be independent, so this condition is not a register hazard.	

Move Conditional and Move CC Latencies

In each of the following cases the stall condition occurs when the same register is used in both instructions.

Table D-21. Move Conditional and Move CC Latencies

Description	Example(s) <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
dreg = CC if CC dreg = **dreg**	1 1	< 1 > **R0** = CC; < 1 + 1 > if CC R1 = **R0**;
if CC **dreg** = dreg multiply/video op using **dreg** as src	1 1 + 1 1 1 + 1	< 1 > if CC **R0** = R1; < 1 + 1 > R2.H = R1.L * **R0**.H; < 1 > if CC **R1** = R3; < 1 + 1 > SAA (R3:2, **R1**:0);
if CC **dreg** = preg math op using **dreg** as src	1 1 1 1	< 1 > if CC **R0** = P0; < 1 + 1 > R2 = R1 + **R0**; < 1 > if CC **R3** = P1; < 1 + 1 > SAA (**R3**:2, R1:0);
dreg = CC math op using **dreg** as src	1 1 1 1	< 1 > **R0** = CC; < 1 + 2 > R2.H = R1.L * **R0**.H; < 1 > **R1** = CC; < 1 + 2 > SAA (R3:2, **R1**:0);
dreg = CC CC = **dreg**	1 1	< 1 > **R0** = CC; < 1 + 2 > CC = **R0**;
if CC **preg** = dpreg any op using **preg**	1 1 + 4	< 1 > if CC **P0** = R1; < 1 + 3 > R4 = **P0**;
if CC **dreg** = dpreg CC = **dreg**	1 1	< 1 > if CC **R0** = R1; < 1 + 1 > CC = **R0**;

Loop Setup Latencies

Table D-22. Loop Setup Latencies

Description	Example <cycles + stalls > instruction	
	BF532	BF535
loop setup loop setup with same LC	1 1 + 6	< 1 > LSETUP (top1, bottom1) LC0 = P0; < 1 + 1 > LSETUP (top2, bottom2) LC0 = P1;
modification of LT or LB loop setup with same loop registers	1 1 + 9	< 1 > LT0 = [SP++]; < 1 + 3 > LSETUP (top, bottom) LC0 = P0;
loop setup with LC0 and LC0 != 0 any processor op	1 1	< 1 > LSETUP (top, bottom) LC0 = P0; < 1 + 1 > NOP;
loop setup with LC1 and LC1 != 0 any processor op	1 1	< 1 > LSETUP (top, bottom) LC1 = P0; < 1 + 1 > NOP;
LC0/LC1 reg written to any processor op	1 1 + 9	< 1 > LC0 = R0; < 1 + 4 > NOP;
LT0/LB0 written to and LC0 != 0 any processor op	1 1 + 9	< 1 > LT0 = [SP++]; < 1 + 4 > NOP;
LT1/LB1 written to and LC1 != 0 any processor op	1 1 + 9	< 1 > LB1 = P0; < 1 + 4 > NOP;
kill while loop buffer is being written due to: *interrupt*, *exception*, *NMI*, *emulation* events	0	3-cycle stall

Instructions Within Hardware Loop Latencies

The following stall conditions occur when the listed instruction or condition within a hardware loop results in a 3-cycle stall at the next iteration of the loop.

- Move conditional or POP into any of the LC/LB/LT registers
- Loop set up through the use of the same loop count registers in the first 3 instructions of the loop
- Branch in the first 3 instructions of the loop (JUMP, CALL, conditional branch)
- Interrupt or exception in the first 4 instructions of the loop
- CSYNC or SSYNC
- The inner hardware loop's bottom is strictly within the outer hardware loop's first four instructions.
- If the inner hardware loop's bottom is equal to the outer hardware loop's bottom, a 3-cycle stall applies to each iteration of the inner loop in addition to the 3-cycle stall of the outer loop.
- RTS, RTN, RTE, RTX, RTI
- If the loop's top instruction is not executed in the first iteration of the loop, a one-time 3-cycle stall penalty is incurred at the beginning of the second iteration (for example, a jump into the hardware loop to any instruction but the first).

- None of the above applies to the 10x core. The 10x core stalls when the loop start does not directly follow the LSETUP. This condition causes a one-time 3-cycle stall while the loop buffer is filled at the beginning of the second loop iteration.
- LSETUP to the same loop count register in the shadow of a previous LSETUP is held in D code until the first LSETUP commits.

Instruction Alignment Unit Empty Latencies

If the instruction alignment unit (IAU) is empty of the next instruction, that next instruction incurs a 1-cycle stall while the IAU is being filled. The following conditions can result in an IAU empty stall.

- Instruction cache miss or SRAM fetch miss
- Change of flow to an instruction address aligned across a 64-bit boundary
- The second instruction after a hardware loop is aligned across a 64-bit boundary
- The sixth instruction within a hardware loop is aligned across a 64-bit boundary

Table D-23. Instruction Alignment Unit Empty Latencies

Description	Example(s) <cycles + stalls > instruction	
	BF532	BF535
Move register or POP to I0 or I1 SAA,BYTEOP2P,BYTEOP3P	1 1 + 4	< 1 > I1 = [SP++]; < 1 + 5 > R0 = BYTEOP3P (R1:0, R1:0) (HI);
Move register or POP to I0 or I1 BYTEOP1P/16P/16M, BYTEUNPACK	1 1 + 4	< 1 > I0 = R0; < 1 + 5 > R3 = BYTEOP1P (R3:2, R1:0);

Table D-23. Instruction Alignment Unit Empty Latencies (Cont'd)

Description	Example(s) <cycles + stalls > instruction	
	BF532	BF535
Write to return register (RT[S,N,E,X,I]) return op	1 1 + 4 1 1 + 4	< 1 > RETI = P0; < 1 + 4 > RTI; < 1 > RETS = P3; < 1 + 4 > RTS;
math op video op with RAW data dependency	1 1 + 1	< 1 > **R3** = R2 + R4; < 1 + 1 > SAA (**R3**:2, R1:0);
dreg = search math op using **dreg**	1 1 + 2	< 1 > (R3, **R0**) = SEARCH R1 (LE); < 1 + 2 > R2.H = R1.L * **R0**.H;
core and system MMR access		< 1 + 2 > R0 = [P0]; // P0 = MMR address
L0/B0 = dreg I0 modulo update In general, any length and base dagreg assignment to a dreg followed by the corresponding index dagreg modulo update	1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4	< 1 > **L0** = R0; < 1 + 3 > R1 = [**I0**++]; < 1 > **B1** = R2; < 1 + 3 > **I1** += 4; < 1 > **L2** = R3; < 1 + 3 > R4 = [**I2**++M2]; < 1 > **B3** = R5; < 1 + 3 > **I3** += M2;

L1 Data Memory Stalls

L1 data memory (DM) stalls are incurred through reading or writing from L1 data memory. Accesses can either be direct (to or from DM SRAM) or indirect (to or from DM cache). Some of these stalls are multicycle instruction conditions (they occur as a result of a specific instruction).

Some stalls are latency conditions (they occur only when the two offending instructions are too close). The specifics are described in each entry. The following memory configurations apply to the ADSP-BF535 processor. Note that the causal factors in offending instructions and the stall consequences appear in **bold** type.

Minibank Access Collision

This section describes the following stalls.

- SRAM access
- Cache access

SRAM Access (1-Cycle Stall)

This stall can occur only when an instruction accesses a bank configured as SRAM. The memory regions associated with SRAM banks are calculated when an offset is added to the value of `SRAM_BASE_ADDRESS MMR`. The start addresses for banks A and B are:

- Bank A: `(SRAM_BASE_ADDRESS << 22) + 0x000000`
- Bank B: `(SRAM_BASE_ADDRESS << 22) + 0x100000`

The minibanks are contiguous 4096 byte (4KB) chunks within the A and B address space. With two simultaneous accesses (via a multi-issue instruction) to the same minibank, a 1-cycle stall is incurred. For example:

```
(I0 is address 0x001348, I1 is address 0x001994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in the second minibank
in superbank A)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (DAG0) and the second is a store (DAG1), the cycles incurred are seen by the store buffer (see "Store Buffer Overflow" on page D-39). Since the `SRAM_BASE_ADDRESS` value must be 4MB aligned (thus each minibank starts at `0xXXXXX000`), it is easy to determine if two addresses are going to collide in a minibank. If `((addr1>>12)==(addr2>>12))`, a collision occurs.

Cache Access (1-Cycle Stall)

This stall can occur only when one or both banks are configured as cache.

Only One Bank is Configured as Cache

In this case, data memory accesses are always cached to the same super-bank, so you have to determine only the cache minibank. You must first find out how much data bank memory is modeled in the implementation of the Blackfin processor that you are using.

The standard Blackfin processor architecture model is 16KB, thus four 4KB minibanks. In this case you have to look at only bits 13 and 12 of the address to see what minibank the data memory access is cached to.

Table D-24. Minibanks Selected for 16KB of Data Bank Memory

Addr[13:12]	Minibank Selected
00	minibank 1 (0x0000–0x1000)
01	minibank 2 (0x1000–0x2000)
10	minibank 3 (0x2000–0x3000)
11	minibank 4 (0x3000–0x4000)

Every time the available bank memory is doubled, another bit must be used. For example, if an implementation of Blackfin processor architecture has 32KB of data bank memory (eight 4-KB memory banks), bits 14, 13, and 12 must be used.

Table D-25. Minibanks Selected for 32KB of Data Bank Memory

Addr[14:12]	Minibank Selected
000	minibank 1 (0x0000–0x1000)
001	minibank 2 (0x1000–0x2000)
010	minibank 3 (0x2000–0x3000)

Table D-25. Minibanks Selected for 32KB of Data Bank Memory

Addr[14:12]	Minibank Selected
011	minibank 4 (0x3000–0x4000)
100	minibank 5 (0x4000–0x5000)
101	minibank 6 (0x5000–0x6000)
110	minibank 7 (0x6000–0x7000)
111	minibank 8 (0x7000–0x8000)

For simplicity, this document assumes the standard 16KB data memory model. If the addresses in a dual memory access (multi-issue) instruction is cached to the same minibank, a 1-cycle stall is incurred.

```
(I0 is address 0x002348, I1 is address 0x002994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in minibank 3)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (DAG0) and the second is a store (DAG1), the cycles incurred are seen by the store buffer (see "Store Buffer Overflow" on page D-39). If `(Addr1[13:12]== Addr2[13:12])`, a collision occurs.

Both Banks Are Configured as Cache

If both banks are cacheable, you must determine which superbank the accesses are cached to (in addition to the minibank) to determine whether a stall exists. This information depends on the value of the `DCBS` bit of the `DMEM_CONTROL MMR`. If `DCBS` is `1`, address bit 23 is used as bank select. If `DCBS` is `0`, address bit 14 is used as bank select.

(Note that these values are used for the 16KB implementation of Blackfin processor data memory). Refer to "Cache Access" for details about how to determine the minibank. The following table assumes that `DCBS` is `0`.

Table D-26. Superbank, Minibank Selected When DCBS is 0

Addr[14:12]	Superbank, Minibank Selected
000	superbank A, minibank 1 (0x0000–0x1000)
001	superbank A, minibank 2 (0x1000–0x2000)
010	superbank A, minibank 3 (0x2000–0x3000)
011	superbank A, minibank 4 (0x3000–0x4000)
100	superbank B, minibank 1 (0x0000–0x1000)
101	superbank B, minibank 2 (0x1000–0x2000)
110	superbank B, minibank 3 (0x2000–0x3000)
111	superbank B, minibank 4 (0x3000–0x4000)

If the addresses in a dual memory access (multi-issue) instruction is cached to the same superbank and minibank, a 1-cycle stall is incurred.

```
(I0 is address 0x002348, I1 is address 0x002994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in minibank 3)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (`DAG0`) and the second is a store (`DAG1`), the cycles incurred are seen by the store buffer (see "Store Buffer Overflow" on page D-39).

When `DCBS` is `0` and `(Addr1[14:12]== Addr2[14:12])`, a collision occurs. When `DCBS` is `1` and `(Addr1[23,13:12]== Addr2[23,13:12])`, a collision occurs.

MMR Access

A read from any MMR space (on-core and off-core) results in a 2-cycle stall because the Blackfin processor architecture must wait for acknowledgement from the peripherals mapped to the MMRs being accessed.

```
(I0 contains an address between 0xFFC00000 and 0xFFE00000)
R2 = [I0++]; (In Supervisor Mode)
<2 cycle stall>
```

System Minibank Access Collision

A system access occurs when some external device, such as another processor in a multiple core system, accesses Blackfin processor architecture L1 memory. Whenever the system accesses a minibank currently being accessed by the core, a **<1-cycle stall>** is incurred because system memory accesses have higher priority than core accesses.

Store Buffer Overflow

The store buffer is a 5-entry FIFO that manages Blackfin processor instruction stores to L1 and L2 memory. All instruction stores must go through the store buffer. Thus, if the buffer is full, the Blackfin processor stalls until the FIFO moves forward and a space is freed.

The earliest time that a store can leave the buffer is 4 instructions (not cycles necessarily) after it was entered. Consequently, under ideal circumstances a continuous series of stores will take up 4 out of the 5 slots in the store buffer. If only one of the stores is delayed by an extra cycle, no penalty is imposed as the store buffer has 5 slots. Many scenarios can cause the store buffer to become full. To account for them, you must keep track of the proximity of stores and how many cycles they each take.

If a multicycle store is required, you must be sure that it is not followed too closely by other stores as they may become backed up. Here is a list of multicycle stores:

- Stores to non-cacheable memory (for example, MMR space)
- Stores to external L2 memory (memory addressed beyond L1 SRAM)
- Minibank conflict where the store is from DAG1 (the second access in a load/store multi-issue instruction—see "Minibank Access Collision" on page D-35)

Store Buffer Load Collision

This section describes cases where a load access collides with a pending store access in the store buffer because they have the same address (refer to section "Store Buffer Overflow" on page D-39 for a description of the store buffer).

Load/Store Size Mismatch

If the load access's size (8 bit, 16 bit, 32 bit) is different from that of the store access, the store buffer must be flushed before the load can be carried out. The stall time depends on how many stores are currently in the buffer and how long they each take to complete.

```
W [P0] = R0;
<N cycle stall as the buffer is flushed>
R1 = B [P0];
```

Store Data Not Ready

The data portion of a store does not necessarily have to be ready when it is entered into the store buffer. Store data coming from the dagregs and pregs has no delay, but all other store data is delayed by 3 instructions. If a load access collides with a store whose data is not ready, the Blackfin processor stalls for 3 cycles.

```
W [P0] = R0;                [P0] = P3;
<3 cycles>                  <0 cycles>
R1 = W [P0];                R1 = [P0];
```

Instruction Groups

All instruction group members conform to naming conventions used in the *Blackfin Processor Instruction Set Reference*. Instruction groups described are not necessarily mutually exclusive in that the same instruction can belong to multiple groups.

Table D-27. Math Ops Instruction Groups

Math Ops		
Video Ops	**Mult Ops**	**ALU Ops**
Video Pixel Ops	Vector Multiply	Logical Ops
	32-bit Multiply	Bit Ops
	Vector MAC	Shift/Rotate Ops
		Arithmetic Ops (except Mult)
		Vector Ops (except Mult/MAC)

Table D-28. Areg2Dreg Ops Instruction Groups

Areg2Dreg Ops		
MAC to half reg	MAC to data reg	Vector Multiply
RND12	RND20	Add on Sign
Modify – Increment, only this case: `[dreg|dreg_hi|dreg_lo] = (A0 += A1);`		

Register Groups

Table D-29. Allreg Register Groups

allreg			
dreg	**preg**	**sysreg**	**dagreg**
R0	P0	ASTAT	I0
R1	P1	RETS	I1
R2	P2	RETX	I2
R3	P3	RETI	I3
R4	P4	RETN	M0
R5	P5	RETE	M1
R6	FP	LC0	M2
R7	SP	LT0	M3
statbits	accreg	LB0	L0
ASTAT [0]: AZ	A0	LC1	L1
ASTAT [1]: AN	A0.x	LT1	L2
ASTAT [2]: AC	A0.w	LB1	L3
ASTAT [3]: AV0	A1	CYCLES	B0
ASTAT [4]: AV1	A1.x	CYCLES2	B1
ASTAT [5]: CC	A1.w	SEQSTAT	B2
ASTAT [6]: AQ		SYSCFG	B3

Compiled Simulation

A traditional simulator decodes and interprets one instruction at a time. Each executed instruction often requires repeated decoding. Compiled simulation removes the overhead of having to decode each instruction repeatedly.

In VisualDSP++ 3.5, compiled simulation had required that you first build a compiled simulation `.EXE` file from a `.DXE` program file and then load and execute the program in a compiled simulation debug target.

In VisualDSP++ 4.0, the intermediate step is no longer required. You can load the `.DXE` program directly into the compiled simulation debug target. When you run, the debug target compiles the processor code into native code and executes the native code.

Specifying a Session for Compiled Simulation

(Blackfin processsors only). You must configure the debug session for compiled simulation.

1. From the VisualDSP++ **Session** menu, choose **New Session**. The **New Session** dialog box appears.
2. In **Debug target**, select **Blackfin Family Compiled Simulator**.
3. In **Platform**, select **Blackfin Family Compiled Simulator**.
4. In **Processor**, select **ADSP-BF535**, **ADSP-BF531**, **ADSP-BF532**, or **ADSP-BF533**.
5. In **Session name**, enter a name for this session.
6. Click **OK**.

I INDEX

Numerics

3-D waterfall plots, *See* waterfall plots

A

abbreviations
  Pipeline Viewer messages (ADSP-BF535), D-17
aborts
  Pipeline Viewer for ADSP-TS101, C-6
  Pipeline Viewer for ADSP-TS20x, C-17
About VisualDSP++ dialog box, A-65
access to ADSP-21065L 9th column Even Address event, B-7
.ACH files, A-25
ActiveX
  script support, 2-48
adding
  files to projects, 1-23
address bar
  Disassembly windows, 2-52
  memory window, 2-73
ADSP-21x6x processors
  reporting anomalies, B-2
anomalies
  ADSP-2116x DSPs, B-10
  ADSP-21x6x processors, B-2
  shadow write FIFO, B-2
  short word, B-4
  SIMD FIFO, B-3
Anomalies submenu (SHARC), B-2
  Shadow Write, B-2
  SIMD FIFO, B-3
anomaly events, recording, B-7
API
  defined, A-2
archiver
  about, 1-41
.ASM files, 2-24, A-24
assembler
  about, 1-30
  file associations for tools, 2-24
  input files, 2-24
  options, 1-30
  terms, 1-30
assembling
  language files into object files, 1-30
assembly instructions
  profiling statistics, 2-70
auto-completion
  scripts, 2-49
automatic file placement, 2-24
automation
  Image Viewer, 2-126
Automation API
  access to, 2-48

B

background telemetry channels, *See* BTCs
batch builds, 1-56

batches
  building, 1-56
Blackfin processors
  peripherals supported in simulators, D-2
.BNM files, A-25
bookmarks
  editor windows, 2-30
  Help, A-56
  keyboard shortcuts, A-35
  Output window, 2-39
Boolean operators
  searching Help, A-63
boot
  kernel, 1-42
  loading or booting, B-12
  options, B-11
booting, 1-43
  simulating, 1-20
boot-loadable files, B-12
boot loading, 1-43
break condition
  defined, A-2
breakpoints
  about, 3-3
  conditional, 3-14
  Disassembly windows, 2-54
  hardware, 3-3
  icons, 3-13
  keyboard shortcuts, A-37
  MP sessions, 3-11
  symbols, 3-13
  unconditional, 3-14
  using, 3-13
BTC_MAP_ENTRY_ASM macro, 2-79
BTC_MAP_ENTRY macro, 2-79
BTC Memory window
  about, 2-81
  right-click menu, 2-84
BTCs
  background telemetry channels, 2-78
  BTC Memory window, 2-81
  changing BTC priority, 2-81
  channel definitions, 2-78
  defining channels, 2-78
  list of defined channels, 2-83
  streams, 3-17
building projects
  about, 1-54
  keyboard shortcuts, A-36
build options
  about, 1-55
  custom, 1-55
  files, 1-25
  individual file, 1-55
  projects, 1-25
  project wide, 1-55
  specifying, 1-24
Build page
  about, 2-38
build status
  viewing, 2-38
build type, *See* configuration
buttons
  appearance on toolbars, 2-9
  built-in toolbars, 2-7
  standard Windows functions, A-75

C

cache events, 2-99
cache events log
  stepping, 2-101
cache events log file, 2-101
cache hits, 2-100
cache misses, 2-100
cache thrashing, 2-100

Cache Viewer
about, 2-99
Address View page, 2-108
cache events log file, 2-101
Configuration page, 2-102
Detailed View page, 2-103
Histogram page, 2-107
History page, 2-104
Performance page, 2-106
right-click menu, 2-101
cache ways, 2-100
Call Stack window, 2-72
.C files, 2-24, A-24
channels
BTC, 2-79
definition, A-3
CLKOUT pin, configuring, B-10
clock doubling, ADSP-21161 processors, B-10
code
development tools, 1-2
code analysis tools, 3-7
code development tools
list of, 1-27
COFF
defined, A-3
colors
indicating focus, 3-7
Output window, 2-38
plots, 3-25
command-line parameters
idde.exe, A-38
commands
control menu, 2-5
DOS, 1-57
program execution, 3-11
single-stepping, 3-11
stepping, 3-11
toolbar buttons, A-27
user tools, 2-12
comments
rules for, A-47
start and stop strings, A-47
compiled simulation, D-43
preparing a program from an existing .DXE file, D-43
compiler, 1-28
file associations for tools, 2-24
input files, 2-24
options, 1-28, 1-31
suppressing warnings and remarks, 2-44
compiling
C programs, 1-28
C++ programs, 1-28
conditional breakpoints, 3-14
configurations
customized, 1-54
Debug, 1-53
project, 1-53
Release, 1-53
Configurator, VisualDSP++, 3-4
Configure Simulator Event dialog box, B-8
configuring
Cache Viewer, 2-102
plots, 2-121
Console page
about, 2-38
constellation plots
about, 3-22
context switch
defined, A-4
control menu, 2-4, 2-5
.CPP files, 2-24, A-24
C programs, compiling, 1-28
C++ programs, compiling, 1-28
creating
files to add to your project, 1-23
new plot windows, 2-121
critical region
defined, A-4

CROSSCORE
  defined, A-4
C++ run-time libraries, 1-29
current program counter value
  ADSP-TS101 DSPs, C-9
  ADSP-TS20x DSPs, C-21
custom build
  options, 1-25
custom-build options, 1-55
customizing
  Output window, 2-46
  plot windows, 2-122
  toolbars, 2-8
custom register windows, 2-89
.CXX files, 2-24, A-24

D

data
  files, 1-30
  I/O simulation, 3-16
  plotting, 3-18
  simulating transfers, 1-20
data collection
  methods, 3-17
data logging
  displaying status in plot windows, 2-117
data sets, 2-121
  definition, 2-121
data streams
  purpose, 3-16
data structures, 1-15
.DAT files, A-24
Debug configuration, 1-53
debugging
  IDDE features, 1-5
  multiple processors, 3-4
  overview of, 1-20
  programs, 1-26
  VisualDSP++ features, 1-5
debugging windows
  control menu, 2-6
  list of, 2-50
debug sessions, 1-20
  defined, A-5
  definition, 3-2
  management, 3-3
  managing, 3-3
  multiple, 3-3
  selecting a new session at startup, 3-10
  selection at startup, 3-10
  setting up, 3-2
  specifying, A-38
  specifying for multiprocessing, 3-4
  switching, 3-3
  types, 3-2
  viewing list of, 3-3
debug targets
  defined, A-5
  specifying, 3-2
debug targets, *See* targets
demoting
  error messages, 2-42
dependencies, project, 1-55
device drivers
  defined, A-6
Disassembly windows, 2-53
  address bar, 2-52
  examples, 2-52
  features, 2-53, 2-54
  going to an address, 2-54
  invoking, 2-52
  opening multiple, 2-53
  pipeline stages, 2-56
  right-click menu, 2-55
  symbols, 2-56
discretionary errors, 2-40
discretionary messages, 2-42
.DLB files, 2-24, A-24
.DLO files, A-24

docked windows, A-71, A-72
docking
  toolbars, 2-8
documentation
  printing, A-48
.DOJ files, 1-28, 1-30, 1-31, 2-24, A-24
DOS commands
  running, 1-57
.DPG files, 1-47, A-26
.DPJ files, 1-44, A-24
.DSP files, 2-24, A-25
DSPs - see processors, 1-2
dumping
  memory, 2-73
DWARF-2, 1-27
  defined, A-7
.DXE files, 1-31, A-24
  loading automatically, 1-26

E

editing
  features, 1-3
  files, 1-24
  keyboard shortcuts, A-34
  memory, 2-72
editor files
  comments in, A-47
Editor Tab mode, 2-33
editor windows
  about, 2-28
  bookmarks, 2-30
  breakpoints, 3-13
  Editor Tab mode, 2-33
  expression evaluation, 2-34
  operations, 2-28
  parts of, 2-28
  program icon, 2-5
  right-click menu, 2-35
  source mode vs. mixed mode, 2-31
  switching among, 2-33
  symbols, 2-30
  syntax coloring, 2-30
ELF, 1-27
ELF/DWARF format, A-24
elfloader.exe, 1-42
emulation
  available tools, 1-20
  debug session management, 3-3
  restarting programs, 3-12
  vs. simulation, 3-3
emulation targets, 1-19
emulator
  defined, A-7
  when to use, 1-18
emulator targets
  statistical profiling, 3-8
error messages, 2-50
  demoting, 2-42
  displaying offending code, 2-38
  log file, 2-45, 2-46
  Output window, 2-36, 2-39
  promoting, 2-42
  scrolling in Output window, 2-38
  severity, 2-42
  severity hierarchy, 2-40
  suppressing, 2-42
  syntax, 2-40
errors
  discretionary, 2-40
evaluating
  expressions in editor windows, 2-34
evaluation
  available tools, 1-20
event bit
  defined, A-8

events
cache, 2-99
defined, A-7
Pipeline Viewer, 2-97
thread, 2-113
using data cursor, 2-113
events log
cache, 2-101
Events submenu (SHARC), B-4
Access to ADSP-21065L 9th column Even Address, B-7
FP Denorm, B-4
Short Word Anomaly, B-4
Excel
data files, 3-17
plot window data, 3-17
executable
loading automatically, 1-26
executables
loading, 3-11
execution traces, 2-60, 3-9
.EXE files, A-25
Expert Linker
about, 1-34
overview, 1-34
stack and heap usage, 1-39
window, 1-35
expressions
C expressions, 2-60
context-sensitive evaluation, 2-34
evaluating in editor windows, 2-34
Expressions window, 2-59
memory windows, 2-77
nested, searching Help, A-64
register, 2-60
regular, A-43
tagged, A-46
tracking, 2-76
tracking in memory windows, 2-72
viewing value of, 2-34
Expressions window
about, 2-57
right-click menu, 2-58
valid expressions, 2-59
external interrupts
generating, 3-16
simulating, 1-20
eye diagrams
about, 3-23
example of, 3-23
EZ-ICE target, 3-2
EZ KIT Lite evaluation systems
license status, 1-11
EZ-KIT Lite evaluation systems
as targets, 1-19
flash drivers, 3-28
planning, 1-18

F

fatal errors, 2-40
features
new in VisualDSP++, 1-7
project management, 1-4
VDK, 1-6
VisualDSP++, 1-2
file building options, 1-25
files
.ASM, 2-24
assembler, 1-30
association with tools, 2-24
automatic placement, 2-24
building, 1-56
.C, 2-24
cache events log, 2-101
compiler, 1-28
.CPP, 2-24
.CXX, 2-24
data, 1-30
.DLB, 2-24
.DOJ, 1-31, 2-24

.DPG, 1-47
.DPJ, 1-44
.DSP, 2-24
DSP project, A-24
executable, 1-32
extensions in VisualDSP++, A-24
keyboard shortcuts, A-32, A-33, A-37
language, 1-30
.LDF, 1-30, 1-32, 2-24
linker, 1-31, 1-32
log, 2-45, 2-46
.MAK, 1-49
.MK, 1-49
nested folders in Project window, 2-21
object, 1-31
options, 1-25
overlay, 1-31
placing into folders automatically, 2-21
project, 2-22
project group, 1-47
Project window rules, 1-57
PROM, 1-41, 1-42
.S, 2-24
vdk_config.cpp, 2-26
vdk_config.h, 2-26
VisualDSP_Log.txt, 2-46
.VPS, 2-116, 2-122
file tree, 2-14
icons, 2-14
Project window, 2-14
filling
memory, 2-73
filtering
PC samples, 2-71
finding
regular expressions in find/replace operations, A-43
tagged expressions, A-46
find/replace operations
regular expressions, A-43
Flag IO (FIO) peripheral
Blackfin, D-2
flash algorithm
loading, 3-27
flash devices
changing data, 3-27
flash drivers
loading, 3-27
flash memory
about, 3-27
erasing, 3-27
filling, 3-27
programming, 3-27
resetting, 3-27
Flash Programmer
about, 3-27
flash driver, 3-28
functions, 3-27
user interface, 3-29
window controls, 3-30
Flash Programmer window, 3-29
floating
toolbars, 2-8
floating windows, A-71, A-73
focus
colors, 3-7
definition, 3-6
moving, 2-93
multiprocessor debug session, 2-90
folders
automatic file placement, 2-24
project, 2-21
Project window, 2-14, 2-21
Project window rules, 1-57
font color in Output window, 2-38
FP Denorm, B-4
functions
displaying, 2-67
displaying local variables, 2-62

G

General page, 1-26
global build options, 1-25
glossary, A-2
graphing
  DSP memory, 3-18
grouping
  processors, 2-92
groups
  multiprocessor, 2-92

H

halting
  programs, 3-12
hardware
  specifying, 3-5
hardware breakpoints, 3-3
  about, 3-14
  latency, 3-15
  restrictions, 3-15
  use, 3-14
Hardware Breakpoints dialog box, 3-15
hardware conditions
  simulating, 1-20
header files, 1-30
heaps
  usage in Expert Linker, 1-39
Help
  about, 1-59
  bookmarking, A-56
  context-sensitive, A-52
  copying example code, A-55
  features and operations, A-51
  invoking, A-50
  keyboard shortcuts, A-37
  navigating, A-57
  navigation buttons, A-52
  printing, A-56
  searching, A-59
  window, A-51
.H files, A-24, A-25
.H_# files, A-25
.HPP files, A-24
.HXX files, A-24

I

ICE
  defined, A-9
ICE Test, A-22
icons
  editor windows, 2-30
  Pipeline Viewer events, 2-97
  Project window, 2-17
IDDE, 1-2
  command-line parameters, A-38
  source code control, 1-48
idde.exe
  command-line parameters, A-38
Idle thread
  time spent in, 2-114
images
  displaying, 2-124
Image Viewer window
  automation interface, 2-126
  features, 2-124
  right-click menu, 2-127
  status bar, 2-127
  toolbar, 2-126
instruction groups, Blackfin processors, D-41
instruction latencies, Blackfin processors, D-26
  accumulator to data register, D-27
  instruction alignment unit empty, D-33
  instructions within hardware loops, D-32
  loop setup, D-31
  move conditional and move CC, D-30
  register move, D-28

instructions
Cache Viewer, 2-99
Prefetch, 2-100
instruction timing analysis
ADSP-TS101 DSPs, C-2
ADSP-TS20x DSPs, C-13
interrupts
about, 3-16
defined, A-9
simulating, 1-20
interrupt service routines
defined, A-9
invoking
IDDE with command-line parameters, A-38
I/O
simulating data transfer, 1-20
.IS files, A-24
ISRs
exercising, 1-20

J

JScript
scripting with, 2-48
.JS files, A-25
JTAG emulator, 3-2
breakpoints, 3-13
debug sessions, 3-3
exchanging data without halting, 2-78
platforms, 3-2
project development, 1-18
statistical profiles, 3-8
statistical profiling, 2-63
JTAG interface
exchanging data, 2-78

K

kernel
defined, A-10
kernel, *See* VDK
Kernel page
about, 2-26
keyboard shortcuts, A-32
list, A-32
kills detected messages, Pipeline Viewer (ADSP-BF535), D-16

L

L1 data memory stalls, D-34
cache access (1-cycle stall), D-36
minibank access collision, D-35
MMR access, D-39
SRAM access (1-cycle stall), D-35
store buffer load collision, D-40
store buffer overflow, D-39
system minibank access collision, D-39
latencies, D-22
.LDF files, 1-31, 2-24, A-24
.LDR files, A-25, B-12
legends
in plots, 3-25
librarian
defined, A-10
libraries, C++ run-time, 1-29
library functions
displaying, 2-67
license installation, 1-12
license management, 1-9
license installation, 1-12
license status, 1-11
licensing options, 1-10
product serial numbers, 1-13
product upgrades, 1-13
temporary licenses, 1-11
valid vs. expired licenses, 1-11
licenses
software, 1-11
linear profiling
features, 3-8

Linear Profiling Results window, 2-63, 3-8
  right-click menu, 2-65
line plots
  about, 3-20
linker
  file associations for tools, 2-24
  input files, 2-24
  overview, 1-31
Linker Description File, 1-32
linking, object files, 1-31, 1-32
loader
  about, 1-42
  specifying options, 1-42
  terms, 1-42
loading
  executable programs, 3-11
  programs, 3-11
  scripts, 2-49
  scripts from a shortcut, A-42
Load Sim Loader submenu options, B-11
local build options, 1-25
Locals window
  about, 2-62
  right-click menu, 2-62
Locate button, A-53
log file
  about, 2-45
.LST files, A-25

M

main window
  program icon, 2-5
  status bar, 2-12
makefiles, 1-49
  example of, 1-51
  Output window, 1-51
  rules, 1-50
.MAK files, 1-49, A-26
managing
  debug sessions, 3-3
  licenses, 1-9
  projects, 1-4
  source files, 1-4
manuals
  online, A-48
  printing, A-48
.MAP files, A-25
MATLAB
  data files, 3-17
  plot window data, 3-17
measuring
  performance, 3-8
memory
  displaying an address's value, 2-72
  dumping, 2-53, 2-73
  filling, 2-53, 2-73
  windows, 2-72
memory map
  defining, 1-33
  using Expert Linker, 1-40
Memory Map window, 2-85
memory pool
  defined, A-12
memory segments, 1-14
memory windows
  about, 2-72
  customization, 2-78
  expression tracking, 2-77
  locking columns, 2-72
  number format examples, 2-73
  number formats, 2-73
  right-click menu, 2-75
  tracking an expression, 2-72
  tracking expressions, 2-76
menu bar
  about, 2-6

menus
application menu bar, 2-6
control menu, 2-4, 2-5
right-click, A-70
system, 2-5
title bar right-click, 2-4
messages
demoting to remarks, 2-44
discretionary, 2-42
kills detected (ADSP-BF535), D-16
Pipeline Viewer (ADSP-BF535), D-12, D-17
stalls detected (ADSP-BF535), D-12
suppressing, 2-44
VDK defined, A-12
VisualDSP_Log.txt file, 2-45
Microsoft Script Debugger
enabling, 2-48
mixed mode, 2-31
editor window, 2-32
pipeline symbols, 2-32
.MK files, 1-49, A-26
multicycle behavior, D-22
multicycle instructions, D-22
32-bit multiply, D-23
ADSP-BF535, D-17
call and jump, D-23
conditional branch, D-23
core and system synchronization, D-24
interrupts and emulation, D-25
linkage, D-25
push or pop multiple, D-22
return, D-24
TESTSET, D-25
multiprocessing
focus, 2-93
pinning window to processors, 2-93
setting the focus, 2-93
setting up debug sessions, 3-4
multiprocessor
groups, 2-92
See multiprocessor debug sessions
multiprocessor debug sessions, 3-4
color usage, 3-7
debugging, 3-5
focus, 3-6
focus and pinning, 3-6
managing, 2-90
pinning windows, 3-6
multiprocessor groups
operating on, 2-92
multiprocessor systems
debugging, 3-4, 3-5
Multiprocessor window, 2-90
debugging with, 3-5
Groups page, 2-92
Status page, 2-91
mutiprocessor program execution
stepping, 3-11

N

nested expressions
defining Help searches, A-64
nested folders, 2-21
New Session dialog box, 3-5
No Boot mode, B-10
nodes, Project window, 2-14

O

object files, 1-30
.OBJ files, A-25
online Help, *See* Help
operations
program execution, 3-11
program execution commands, 3-11
optimizing
programs, 2-100

Output window, 2-36
  bookmarks, 2-39
  Build page, 2-38
  capturing messages to log file, 2-45
  Console page, 2-38
    scripting, 2-48
  customization, 2-46
  error messages, 2-39
  loading scripts, 2-47
  makefile errors, 1-51
  parts of, 2-38
  printing, 2-47
  right-click menu, 2-47
  scripts, 2-46
  scrolling previous commands, 2-38
overlays, 1-15, 1-32
overriding
  project-wide options, 1-55
.OVL files, 1-31, A-24

P

PC samples
  filtering, 2-71
performance
  measuring, 3-8
  optimizing, 2-100
peripherals
  Blackfin, simulating, D-1
  limitations in simulation for Blackfin, D-7
  support in simulators for Blackfin, D-2
  Timer (TMR), Blackfin, D-7
  UART for Blackfin, D-7
PGO
  defined, A-14
physical memory
  defining, 1-33
pinning
  definition, 3-6
pipeline
  kill reasons (ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF561), D-20
  kill reasons (ADSP-BF535), D-10
  stage event icons in Pipeline Viewer, 2-97
  stall reasons (ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF561), D-19
  stall reasons (ADSP-BF535), D-9
pipeline stages
  ADSP-TS101 DSPs, C-2
  ADSP-TS20x DSPs, C-14
  Disassembly windows, 2-56
pipeline symbols
  mixed mode, 2-32
Pipeline Viewer
  aborts (ADSP-TS101 DSPs), C-6
  aborts (ADSP-TS20x DSPs), C-17
  current program counter value (ADSP-TS101 DSPs), C-9
  current program counter value (ADSP-TS20x DSPs), C-21
  event details, 2-98
  event icons, 2-97
  message abbreviations (ADSP-BF535), D-17
  pipeline stages (ADSP-TS101 DSPs), C-2
  pipeline stages (ADSP-TS20x DSPs), C-14
  specifying properties, 2-96
  stalls (ADSP-TS101 DSPs), C-3
  stalls (ADSP-TS20x DSPs), C-15
  stepping (ADSP-TS101 DSPs), C-11
  stepping (ADSP-TS20x DSPs), C-23
  window, 2-94, C-21
  window messages (ADSP-BF535), D-12
Pipeline Viewer window, 2-94, C-2
  right-click menu, 2-95

pipelining
  defined, A-13
platforms
  about, A-14
  definition, 1-19
  specifying, 3-2
plots
  buffer capacity, 2-116
  colors, 3-25
  configuring, 2-115
  configuring presentation, 2-122
  constellation, 3-22
  data logging status, 2-117
  data sets, 2-121
  DSP memory, 3-18
  eye diagram, 3-23
  legends, 3-25
  line, 3-20
  presentation options, 2-124
  See also plot windows
  spectrogram, 3-26
  types of, 3-19
  viewing statistics, 2-120
  waterfall, 3-24
  X-Y, 3-21
plot windows, 2-115
  BTC mode, 2-116
  configuring, 2-121
  features, 2-116
  operations in, 2-119
  presentation of, 2-122
  right-click menu, 2-118
  See also plots
  status bar, 2-116
  streams, 3-17
  toolbar, 2-117
  types of, 3-19
  viewing statistics, 2-120
positioning
  windows, A-74
post-build options, 1-56
  command syntax, 1-57
.PP files, A-24
pragmas, 2-42
pre-build options, 1-56
preferences
  load file and advance to main, 1-26
  VisualDSP++ and tool output color, 2-38
Preferences dialog box, 1-26, 2-38
Prefetch instructions, 2-100
printing
  hardware manuals, A-48
  online Help, A-56
  VisualDSP++ manuals, A-48
processor, loading into simulator, B-10
processors
  code development tools, 1-2
product
  serial numbers, 1-13
profile-guided optimization
  defined, A-14
profiling
  about, 3-8
  assembly instructions, 2-70
  defined, A-15
  functions, 2-67
profiling results windows, 2-63
Program Counter (PC) register
  setting, 3-12
program development steps
  adding and editing project source files, 1-23
  build a release version of the project, 1-26
  building a debug version of the project, 1-26
  creating a project, 1-23
  setting project options, 1-23

program execution
  commands, 3-11
  halting, 3-11
  keyboard shortcuts, A-36
program icons, 2-5
programming tips, 1-14
program operations
  breakpoints, 3-13
  hardware breakpoints, 3-14
  restarting programs, 3-12
  selecting a debug session at startup, 3-10
  unconditional & conditional
    breakpoints, 3-14
  using program execution commands,
    3-11
  watchpoints, 3-14
programs
  debugging, 1-26
  optimizing, 2-100
  running, 1-26
program sections, 1-14
program traces, 2-60, 3-9
Project box (showing active project), 1-47
project build
  definition of, 1-54
  specifying options, 1-24
project dependencies
  example of, 1-57
project group files, 1-47
project groups, 1-46
project management
  features, 1-4
Project Options dialog box, 1-14
projects
  build options, 1-25, 1-55
  configurations, 1-53
  customized configurations, 1-54
  debugging, 1-5, 1-20
  defined, A-15
  dependencies, 1-24
  development overview, 1-14
  development stages, 1-17
  files, 2-22
  folders, 2-14
  keyboard shortcuts, A-36
  loading, A-38
  nodes, 2-14
  programming overview, 1-14
  project groups, 1-46
  specifying tool options, 1-45
  subfolders, 2-14
  VisualDSP++, 1-44
project-wide file and tool options, 1-25
Project window, 2-14
  about, 2-14
  files, 2-14
  folders, 2-21
  Kernel page, 1-22, 2-26
  makefiles, 1-49
  nodes, 2-15, 2-17
  Project view, 2-15
  right-click menu, 2-18
  rules, 1-57
  source code control icons, 2-23
Project wizard, 1-23, A-16
PROM files, 1-41, 1-42
promoting
  error messages, 2-42
property pages, 1-23
  definition, 1-14
pull-tabs, A-71

R

register expressions, 2-60
register groups, Blackfin processors, D-42
register windows
  about, 2-86
  custom, 2-89
regression tests, 1-16

regular expressions, A-43
  reference texts, A-46
  repetition and combination characters, A-45
  special characters, A-44
Release configuration, 1-53
remarks, 2-40, 2-42
replace operations
  tagged expressions, A-46
replacing
  tagged expressions, A-46
restarting
  programs, 3-12
right-click menus, A-70
running
  programs, 1-26, 3-12
running to cursor, 3-12

S

SCC, *See* source code control (SCC)
scripting
  about, 2-48
  selecting the language, 2-47
  specifying the language, 2-49
scripts
  auto-completion, 2-49
  examples, 2-49
  issuing from an editor window, A-41
  issuing from command lines, A-39
  issuing from menus, A-40
  issuing from Output window, 2-39, A-39
  issuing from user tools, A-41
  loading, 2-49
  loading from a shortcut, A-42
  loading from Output window, 2-47
  running, A-38
  viewing script command status, A-40
scroll bars
  using, A-71
searches
  normal, A-43
  regular expressions vs. normal, A-43
  special character rules, A-45
searching
  Help, A-59
section
  definition of, 1-32
Select Processor ID submenu options, B-10
semaphores
  defined, A-18
sequences
  special characters, A-45
serial numbers, 1-13
Session List dialog box, 3-10
sessions
  compiled simulation, D-43
sessions, *See* debug sessions
setting
  custom build options, 1-25
.S files, 2-24, A-24
.S_# files, A-25
shadow write FIFO anomaly, B-2
shortcut keys, *See* keyboard shortcuts
short word anomaly, B-4
SIMD FIFO, B-3
simulating
  booting, 1-20
  data transfers, 1-20
  external interrupts, 1-20
  hardware, 1-20
  input/output data, 3-16
  random interrupts, 1-20
  SHARC processors, B-1
  TigerSHARC processors, C-1

simulation
  Blackfin processors, D-1
  compiled (Blackfin), D-43
  debug session management, 3-3
  limitations of software models (Blackfin), D-7
  loading a processor, B-10
  options, B-1
  platforms, 1-19
  restarting programs, 3-12
  SPI in slave mode, B-13
  Timer (TMR) peripheral, Blackfin, D-7
  UART peripheral in Blackfin, D-7
  vs. emulation, 3-3
simulation targets, 1-18
simulator
  ADSP-21065L processors, B-4, B-7
  ADSP-2106x processors, B-4, B-10
  ADSP-21161 processors, B-10, B-13
  ADSP-2116x processors, B-2, B-3, B-10
  ADSP-21x6x processors, B-2, B-7, B-11
  ADSP-TS101 processors, C-1
  ADSP-TS20x processors, C-13
  Blackfin peripheral support, D-2
  instruction timing analysis (ADSP-TS101 processors), C-2
  instruction timing analysis (ADSP-TS20x processors), C-13
  sampling PC, 3-8
simulator instruction timing analysis
  ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF561 processors, D-19
  ADSP-BF535 processors, D-9
simulators
  as targets, 1-18
Simulator submenu options (SHARC), B-10
simulator targets
  linear profiling, 3-8
single-stepping, available commands, 3-11
.SM files, 1-31, A-24
software
  serial numbers, 1-13
  upgrades, 1-13
source code control (SCC)
  about, 1-48
  Project window symbols, 2-23
source files
  comments in, A-47
  editing features, 1-3
  management, 1-4
  project, 2-22
  source code control, 1-48
source mode, 2-31
source mode, editor windows, 2-31
spectrogram plots, 3-26
  example of, 3-26
  FFT output, 3-26
SPI simulation in slave mode, B-13
splitter
  about, 1-41
  specifying options, 1-41
stacks
  usage in Expert Linker, 1-39
stack windows, 2-89
stalls, Pipeline Viewer
  ADSP-TS101 DSPs, C-3
  ADSP-TS20x DSPs, C-15
stalls detected messages (ADSP-BF535), D-12
standard output
  viewing, 2-38
startup code, 2-14
statistical profiling
  features, 3-8
Statistical Profiling Results window, 2-63, 3-8
  right-click menu, 2-65

statistics
viewing in plots, 2-120
status bar
examples, 2-12
Image Viewer window, 2-127
information types available, 2-12
plot windows, 2-116
status icons
editor window, 2-30
Pipeline Viewer, 2-97
status messages, log file, 2-45
stepping
available commands, 3-11, 3-12
cache events log, 2-101
Pipeline Viewer
ADSP-TS101 DSPs, C-11
ADSP-TS20x DSPs, C-23
stepping into instructions, 3-12
stepping out of instructions, 3-12
stepping over instructions, 3-12
.STK files, A-25
streams
simulating data I/O, 3-16
used with interrupts, 3-16
subfolders, project tree, 2-14
suppressing
error messages, 2-42
switching
among editor windows, 2-33
symbols
Disassembly window, 2-56
editor window, 2-30
syntax coloring
editor windows, 2-30
system configurator
VDK defined, A-21
system menu, 2-5

T

tagged expressions
finding and replacing, A-46
Target Load window, 2-114
targets, 1-18
about, 1-18
emulators, 1-19
EZ-KIT Lite evaluation systems, 1-19
platforms, 1-19
simulation, 1-18, 1-20
target status messages
Output window, 2-38
.TC8 files, A-25
Tcl, A-39, A-40
interpreter, 2-48
menu issuance, A-40
.TCL files, A-25
terms
VisualDSP++, A-2
text
locating using regular expressions, A-43
text manipulation
keyboard shortcuts, A-34
text selection
keyboard shortcuts, A-34
third-party tools, 1-2
threads, 2-109
defined, A-21
idle, 2-114
status, 2-109, 2-113
tracing, 2-112
Timer (TMR) peripheral
Blackfin, D-7
title bar, 2-4
components, 2-3
indicating focus, 3-6
TMR (see Timer peripheral), D-7
toggling
breakpoints, 3-13

toolbar
list of buttons, A-27
plot windows, 2-117
toolbar buttons
list of, A-27
toolbars, 2-7
built-in, 2-7
button appearance, 2-9
customization, 2-8
docked vs. floating, 2-8
Image Viewer window, 2-126
shape, 2-11
tools
code analysis, 3-7
code development, 1-2
command-line invocation, 1-45
context-sensitive Help, 2-8
documentation, A-48
file associations, 2-24
options, 1-25
third-party, 1-2
user configured, 2-12
Tools menu, user tools, 2-12
traces, 2-61
about, 3-9
Trace windows, 2-60, 3-9
tracking
expressions, 2-76, 2-77
.TXT files, A-25

U

UART peripheral in Blackfin, D-7
unconditional breakpoints, 3-14
Universal Asynchronous
Receiver/Transmitter (UART)
peripheral
Blackfin, D-7
user interface, parts of, 2-2
user tools, 2-12
utilities
ICE Test, A-22

V

variables
global vs. local, 2-59
.VBS files, A-25
VDK
about, 1-22
defined, A-22
features, 1-6
Kernel page, 2-26
load, 2-114
overview of, 1-6
Project window, 2-26
VDK State History window, 2-111
VDK Status window, 2-109
vdk_config.cpp, 2-26
vdk_config.h, 2-26
.VDK files, A-25
VDK State History window, 2-111
right-click menu, 2-113
VDK Status window, 2-109
viewing
memory, 2-72
Visual Basic
scripting with, 2-48
VisualDSP++
Automation API, 2-48
control menu, 2-5
debugging facilities, 1-21
debugging features, 1-5
editing features, 1-3
editor windows, 2-28
environment, 1-2
features, 1-2
file association for tools, 2-24
files, A-24
glossary, A-2
Help system, 1-59, A-51

IDDE, 1-2
kernel, 1-22
keyboard shortcuts, A-32
licenses, 1-9
log file, 2-45
main window parts, 2-3
menu bar, 2-6
new features, 1-7
overview of, 1-1
printing documentation, A-48
product serial numbers, 1-13
product upgrades, 1-13
programming overview, 1-14
project development, 1-17
projects, 1-44
Project window, 2-15
source code control, 1-48
source file editing features, 1-3
toolbar buttons, A-27
user interface, 2-2
VisualDSP++ Configurator, 1-19
describing hardware, 3-4
VisualDSP++ log file
Console page messages, 2-39
VisualDSP_Log.txt, 2-46
script output, 2-48
.VPS files, 2-122

W

warnings, 2-40
promoting to errors, 2-43
watchpoints
about, 3-4, 3-14
MP sessions, 3-11
used with interrupts, 3-16
using, 3-14
waterfall plots
about, 3-24
grid of sampled data, 3-25
rotating, 3-24
ways
cache, 2-100
Windows
standard buttons, A-75
windows
BTC Memory, 2-81
buttons, A-75
Call Stack, 2-72
color indicates focus, 3-7
color used to indicate focus, 3-7
custom register, 2-89
debugging, 2-50
Disassembly, 2-52, 2-54
docked, A-71
Expert Linker, 1-35
Expressions, 2-57
Flash Programmer, 3-29
floating, A-71, A-73
focus while in an MP debug session, 3-6
Help, A-51
Image Viewer, 2-124
indicating focus, 3-6
keyboard shortcuts, A-37
Linear Profiling Results, 2-63, 3-8
Locals, 2-62
MDI, A-70
Memory Map, 2-85
Multiprocessor, 2-90
operating on, A-70
Output, 2-36
parts of the user interface, 2-2
Pipeline Viewer, *See* Pipeline Viewer window
plot, 2-115
profiling results, 2-63
Project, 2-15
pull-tabs, A-71
register, 2-86
rules for positions, A-74
scroll bars, A-71

setting focus, 2-93
stack, 2-89
Statistical Profiling Results, 2-63, 3-8
Target Load, 2-114
Trace, 2-60, 3-9
VDK State History, 2-111
VDK Status, 2-109
wizards
  Project, 1-23, A-16
workspaces
  keyboard shortcuts, A-37

X

X-Y plots
  about, 3-21